RasPi

DESIGN
BUILD
CODE

28

Get hands-on with your Raspberry Pi

+ EMBED PYTHON IN C

DEVELOP WITH PYTHON

Plus Control lights with your Pi

Welcome

Just how much can you do with a Raspberry Pi? I'm not sure we'll ever find out, because people are producing amazing new projects all the time that use the Pi in new and different ways. Take the Pixel Globe project in this issue, then compare it to the tutorial on controlling lights, or the second part of our guide to coding a videogame in FUZE Basic. These projects couldn't be more different, but they all have the Pi at their heart.

One of the things that makes this possible is Python, the ultra-powerful programming language that we take an in-depth look at this issue. With Python, you can ask the Pi to do almost anything – the only real limit is your imagination! Check out our guide for more.

April

Editor

Get inspired

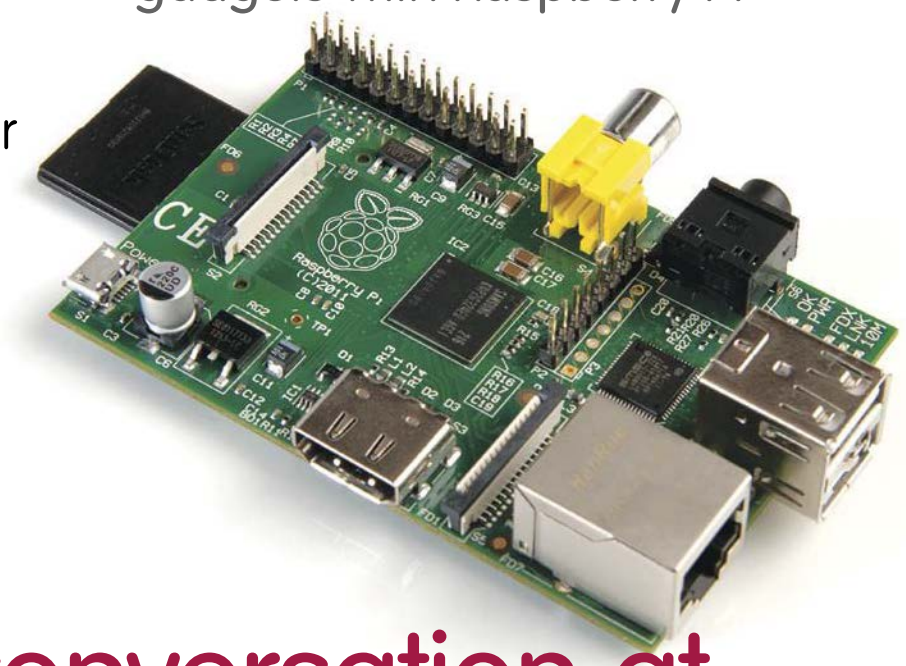
Discover the RasPi community's best projects

Expert advice

Got a question? Get in touch and we'll give you a hand

Easy-to-follow guides

Learn to make and code gadgets with Raspberry Pi



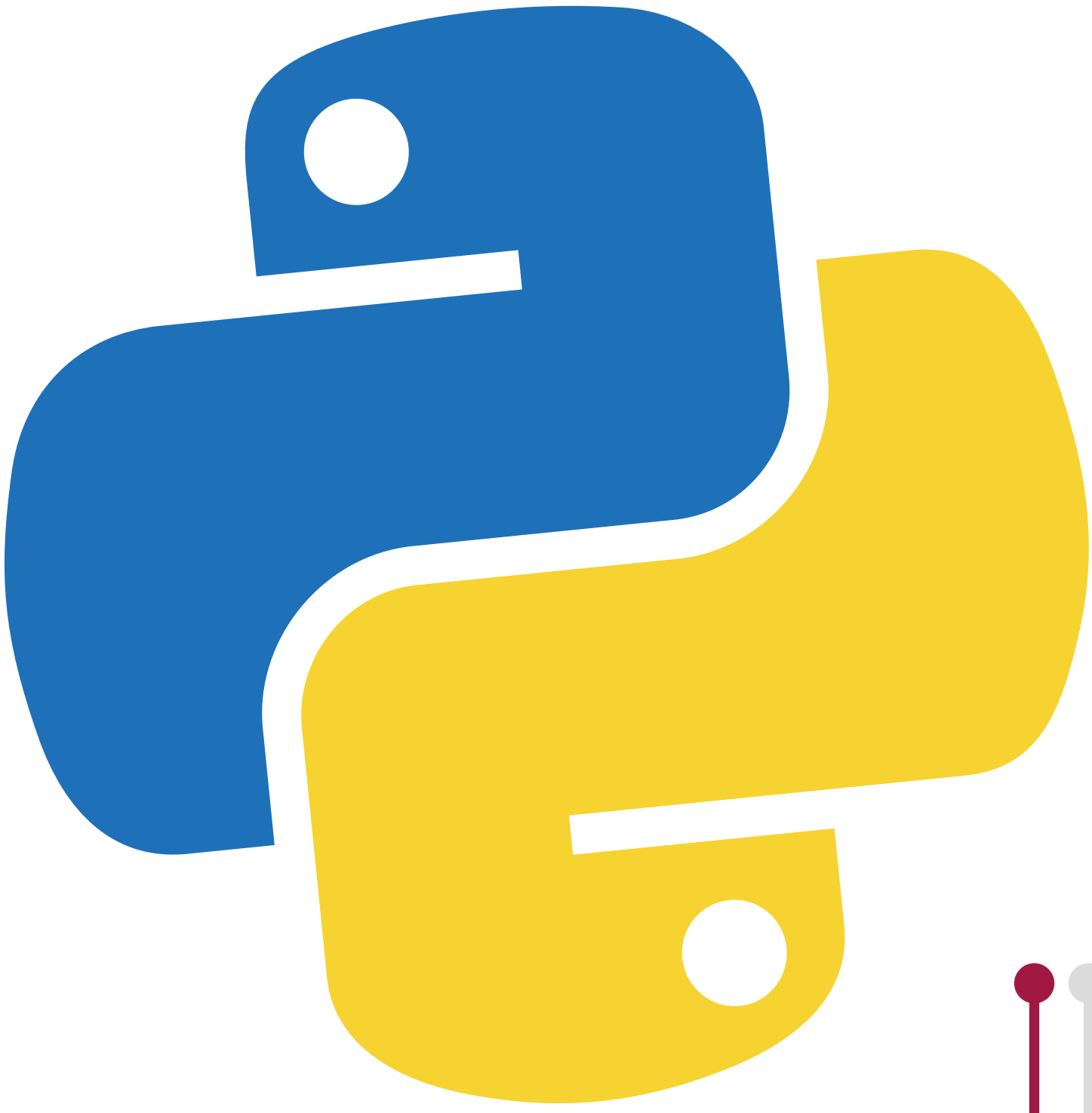
Join the conversation at...

@linuxusermag | Linux User & Developer | RasPi@imagine-publishing.co.uk

Contents

Develop With Python

Python is relied upon by web developers, academic researchers and engineers across the world. Here's how to put your Python skills to professional use



System Administration

Get the most out of Python in handling all of the day-to-day upkeep that keeps your system healthy

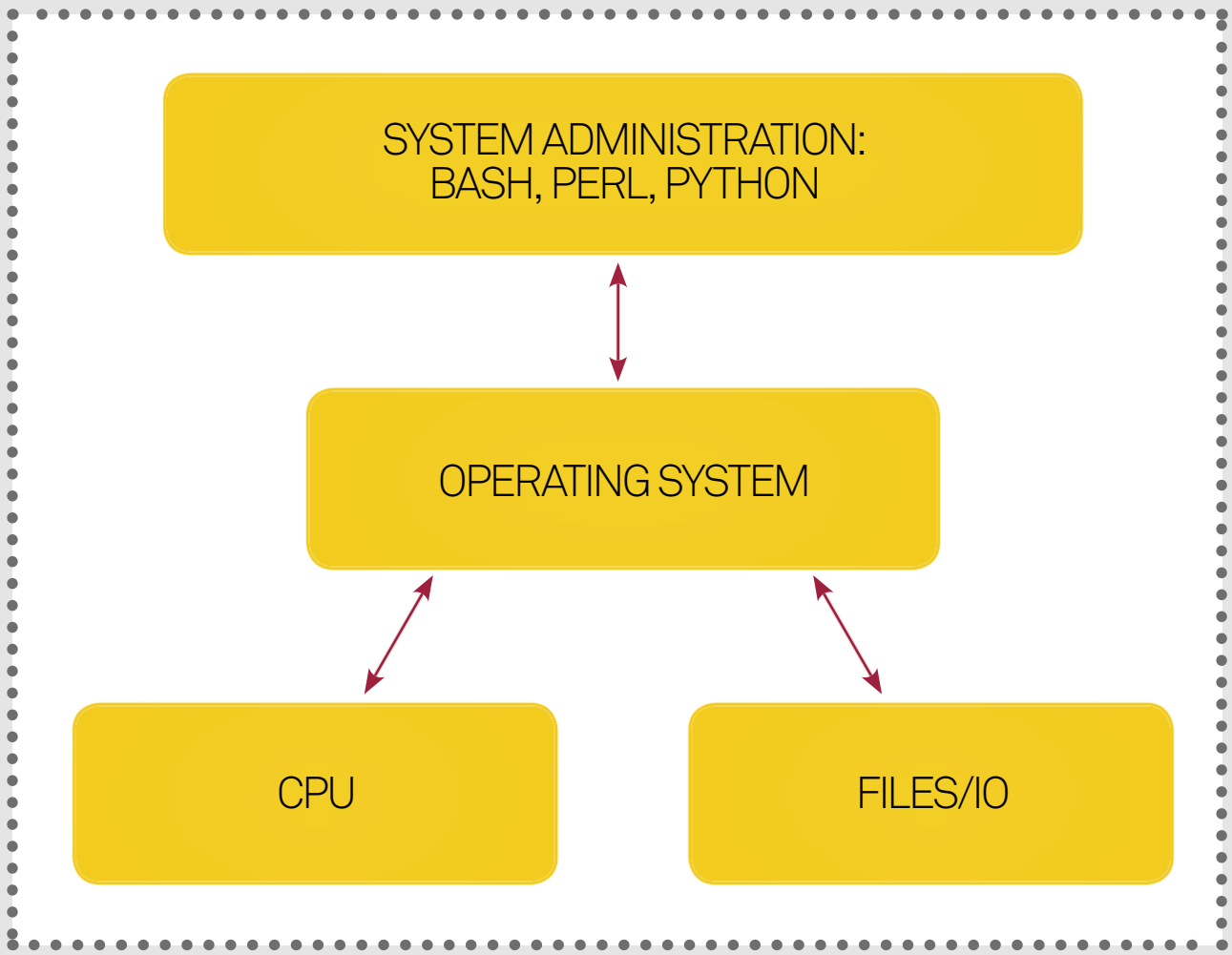
System administration tasks are some of the most annoying things that you need to deal with when you have to maintain your own system. Because of this, system administrators have constantly been trying to find ways to automate these types of tasks to maximise their time. They started with basic shell scripts, and then moved on to various scripting languages. For a long time, Perl had been the language of choice for developing these types of maintenance tools. However, Python is now growing in popularity as the language to use. It has reached the point where most Linux distributions have a Python interpreter included in order to run system scripts, so you shouldn't have any excuse for not writing your own scripts.

Because you will be doing a lot system level work, you will have most need of a couple of key Python modules. The first module is "os". This module provides the bulk of the interfaces to interacting with the underlying system. The usual first step is to look at the environment your script is running in to see what information might exist there to help guide your script. The following code gives you a mapping object where you can interact with the environment variables active right now:

```
import os
os.environ
```

THE PROJECT ESSENTIALS

Pygame Zero
http://pygame-zero.readthedocs.org
Pygame
http://pygame.org
Pip
https://pip.pypa.io/en/stable/
Python 3.2 or later
https://www.python.org



Left Python scripts enable you to instruct and interact with your operating system

You can get a list of the available environment variables with the function **os.environs.keys()**, and then access individual variables with os.environs[k-ey]. These environment variables are used when you spawn a subprocess, as well. So you will want to change values, like the PATH or the current working directory, in order to run these subprocesses correctly. While there is a putenv function that edits these values, it doesn't exist on all systems, so the most beneficial way in the long run is to do this is to edit the values directly within the environs mapping.

Another category of tasks you may want to automate is when working with files. Get the current working directory with:

```
cwd = os.getcwd()
```

You can then get a list of the files in this directory with:

```
os.listdir(cwd)
```

You can move around the filesystem with the function os.chdir(new_

path). Once you've found the file you are interested in, you can open it with os.open() and open it for reading, writing and/or appending. You can then read or write to it with the functions os.read() and os.write(). Once you are all done, you can close the file with os.close().

Running subprocesses from Python

The underlying philosophy of Unix is to build small, specialised programs that do one job extremely well. You then chain these together to build more complex behaviours. There is no reason why you shouldn't use the same philosophy within your Python scripts. There are several utility programs available to use with very little work on your part. The older way of handling this was through using functions like popen() and spawnl() from the os module, but a better way of running other programs is by using the subprocess module instead. You can then launch a program, like ls, by using:

```
import subprocess
subprocess.run(['ls', '-l'])
```

This gives a long file listing for the current directory. The function run() was introduced in Python 3.5 and is the suggested way of handling this. If you have an older version, or need more control, you can use the underlying Popen() function instead. If you want to get the output, you can use:

```
cmd_output = subprocess.run(['ls', '-l'],
stdout=subprocess.PIPE)
```

The variable "cmd_output" is a CompletedProcess object that contains the return code and a string holding the stdout output. It may not be the same way that you are used to, but the methodology is essentially the same.

Scheduling with cron

Once you have your scripts all written up, you may want to schedule them to run automatically without your intervention. On Unix systems, you can have cron run your script on whatever schedule is necessary. The utility crontab -l lists the current contents of your cron file, and crontab -e lets you edit the scheduled jobs that you want cron to run.

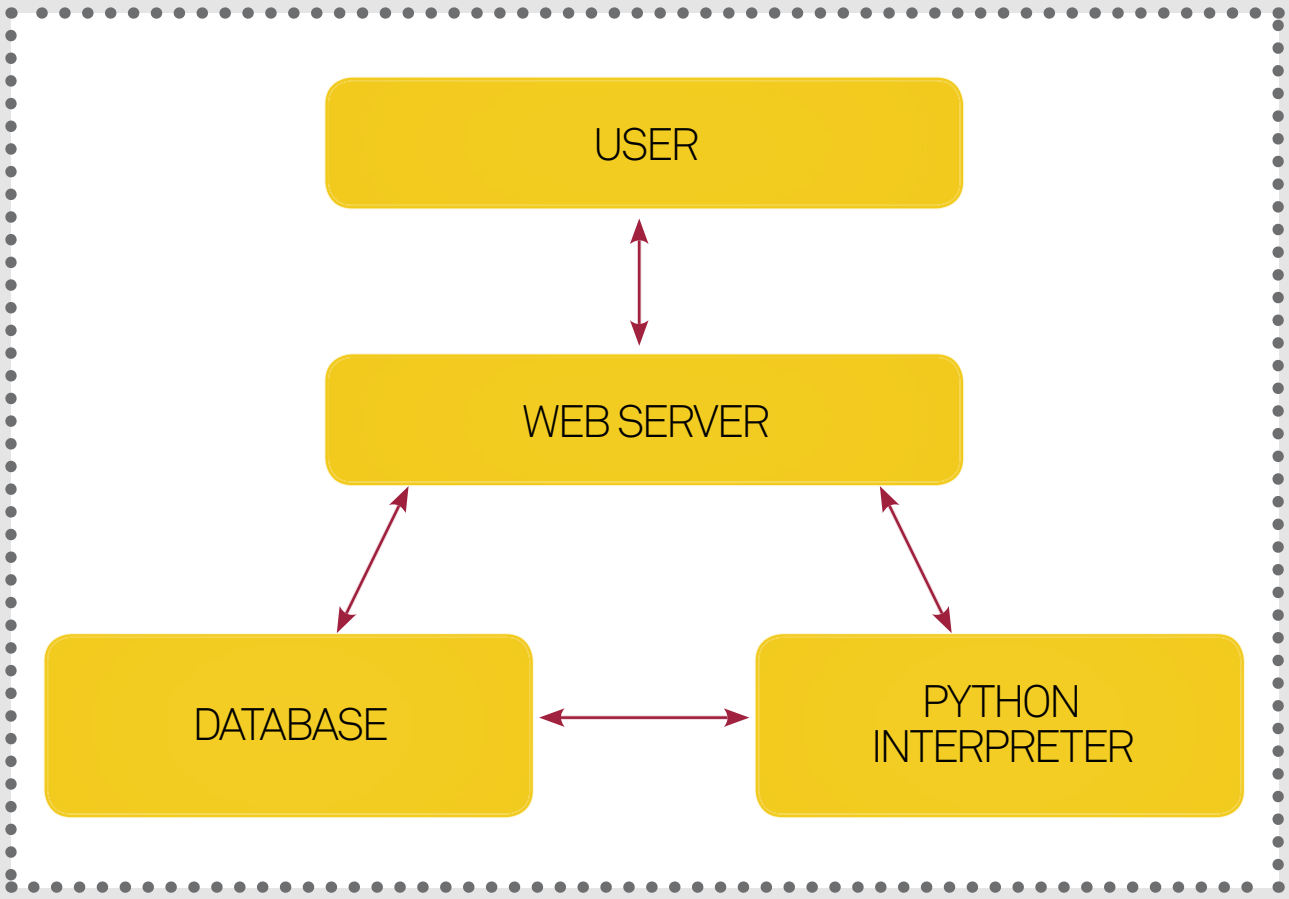
Web Development

Python has several frameworks available for your web development tasks. We will look at some of the more popular ones

With the content and the bulk of the computing hosted on a server, a web application can better ensure a consistent experience for the end user. The popular Django framework provides a complete environment of plugins and works on the DRY principle (Don't Repeat Yourself). Because of this, you should be able to build your web application quickly. Since Django is built on Python, you should be able to install it with sudo pip install Django. Depending on what you want to do with your app, you may need to install a database like MySQL or PostgreSQL to store your application data. There are Django utilities available to automatically generate a starting point for your new project's code:

```
django-admin startproject newsite
```

This command creates a file named "manage.py" and a subdirectory named "newsite". The file "manage.py" contains several utility functions you can use to administer your new application. The new subdirectory contains the files "__init__.py", "settings.py", "urls.py" and "wsgi.py". These files, and the subdirectory they reside in, comprise a Python package that is loaded when your website is



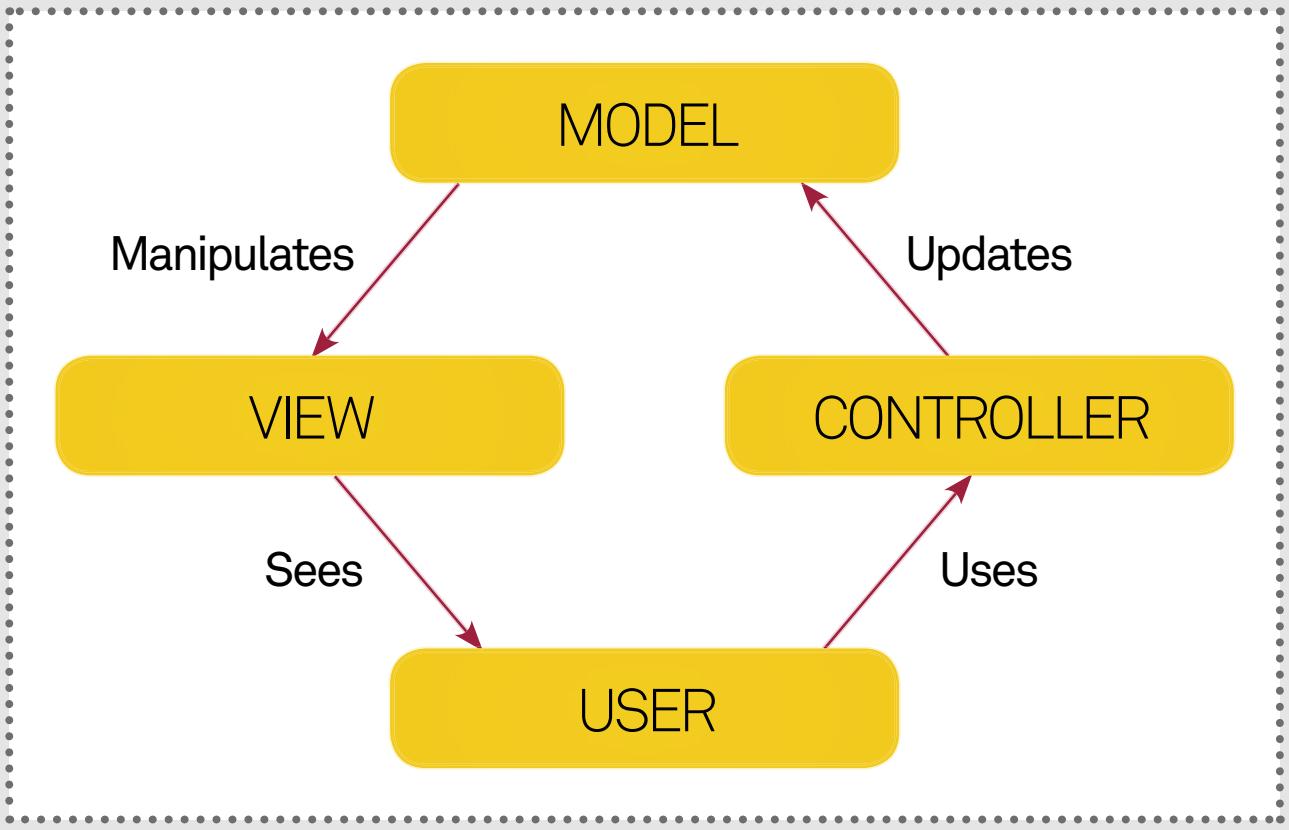
Left Python interpreters work with databases to power a web server

started up. The core configuration for your site can be found in the file "settings.py". The URL declarations, basically a table of contents for your site, are stored in the file "urls.py". The file "wsgi.py" contains an entry point for WSGI-compatible web servers.

Once your application is done, it should be hosted on a properly configured and hardened web server. But this is inconvenient if you are in the process of developing your web application. To help you out, Django has a web server built into the framework. You can start it up by changing directory to the "newsite" project directory and running this command:

```
python manage.py runserver
```

This will start up a server listening to port 8000 on your local machine. Because this built-in server is designed to be used for development, it reloads your Python code for each request, so you don't need to restart the server to see your code changes.



Left The Model-View-Controller architecture is often used for UIs

All of these steps get you to a working project. You are now ready to start developing your applications. Within the "newsite" subdirectory, type:

```
python manage.py startapp newapp
```

This will create a new subdirectory named "newapp", with the files "models.py", "tests.py" and "views.py", among others. The simplest possible view consists of the code:

```
from django.http import HttpResponse
def index(request):
  return HttpResponse("Hello world")
```

This isn't enough to make it available, however. You will also need to create a URLconf for the view. If the file "urls.py" doesn't exist yet, create it and then add the code:

```
from django.conf.urls import url
from . Import views
```

```
urlpatterns = [ url(r‘^$’, views.index,
  name=‘index’), ]
```

Next, get the URL registered within your project with this code:

```
from django.conf.urls import include, url
from django.contrib import admin
urlpatterns = [ url(r‘^newapp/’,
include(‘newapp.urls’)),
      url(r‘^admin’, admin.site.urls), ]
```

This needs to be put in the "urls.py" file for the main project. You can now pull up your newly created application with the URL **http://localhost:8000/newapp/**.

The last part for applications is usually the database. The actual connection details to the database, like the username and password, are contained in the file "settings.py". This connection information is used for all of the applications that exist within the same project. Create the core database tables for your site with:

```
python manage.py migrate
```

For your own applications, you can define the data model you need within the file "models.py". Once the data model is created, you can add your application to the INSTALLED_APPS section of the "settings.py" so that Django knows to include it in any database activity. You initialise it with:

```
python manage.py makemigrations newapp
```

Virtual environments

When you start developing your own applications, you may begin a descent into dependency hell. Several Python packages depend on other Python packages. This is its strength, but also its weakness. Luckily, you have virtualenv available to help tame this jungle. You can create new virtual environments for each of your projects. In this way, you can be sure to capture all of the dependencies for your own package.

Once created, apply these migrations to the database:

```
python manage.py migrate
```

Any time you make changes to your model, you will need to run the makemigrations and migrate steps again.
Once you have your application finished, you can make the move to the final hosting server. Don't forget to check the available code within the Django framework before putting too much work into developing your own.

Other Python Frameworks

While Django is one of the most popular frameworks around for doing web development, it is by no means the only one around. There are several others available that may prove to be a better fit for particular problem domains. For example, if you are looking for a really self-contained framework, you could look at web2py. Everything you need to be able to have a complete system, from databases to web servers to a ticketing system, are included as part of the framework. It is so self-contained that it can even run from a USB drive.

If you need even less of a framework, there are several mini-frameworks that are available. For example, CherryPy is a purely Pythonic multi-threaded web server that you can embed within your own application. This is actually the server included with TurboGears and web2py. A really popular microframework is a project called flask. It includes integrated unit testing support, jinja2 templating and RESTful request dispatching.

One of the oldest frameworks around is zope, now up to version 3. This latest version was renamed BlueBream.

Terminal development environments

When you are in the middle of developing your application, you may need to have several different terminal windows open in order to have a code editor open, a monitor on the server, and potentially somewhere to test and monitor output. If you are doing this on your own machine, this isn't an issue. But if you are working remotely, you should look into using tmux. This can provide a much more robust terminal environment for you to work in.

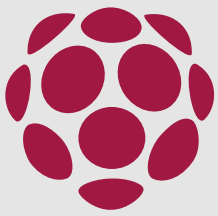
Zope is fairly low-level, however. You may be more interested in looking at some of the other frameworks that are built on top of what is provided by zope. For example, pyramid is a very fast, easy-to-use framework that focuses on the most essential functions required by most web applications. To this end, it provides templating, the serving of static content, mapping of URLs to code, among other functions. It handles this while providing tools for application security.

If you are looking for some ideas, there are several open source projects that have been built using these frameworks, from blogs and forums to ticketing systems. These projects can provide some best-practices when you go to construct your own application.

"To help you out, Django has a web server built into the framework"

Computational science

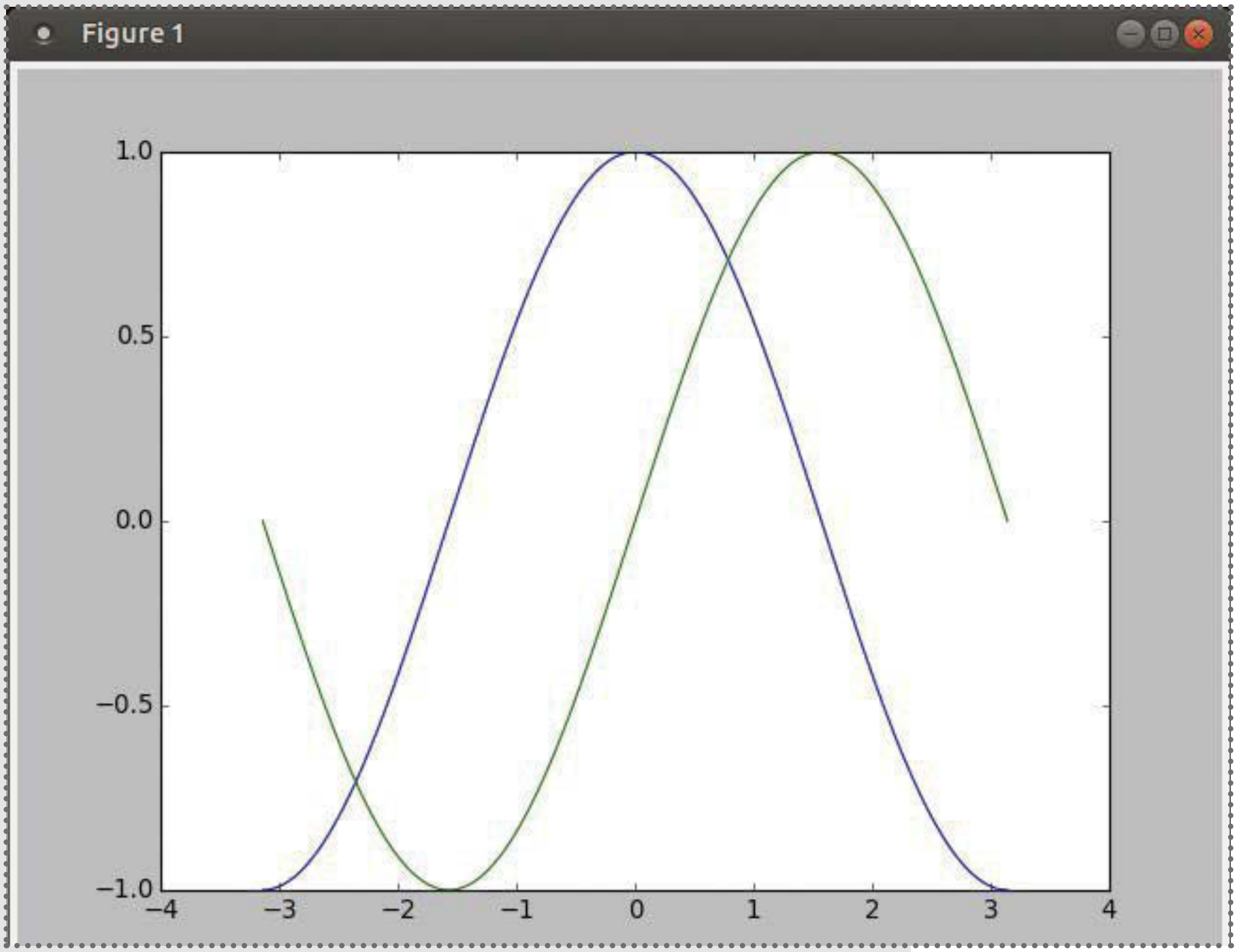
Due to its wide array of packages, Python is fast becoming the go-to language for computational science



Python has become one of the key languages used in science. There is a huge number of packages available to handle almost any task that you may have and, importantly, Python knows what it isn't good at. To deal with this, Python has been designed to easily incorporate code from C or FORTRAN. This way, you can offload any heavy computations to more efficient code.

The core package of most of the scientific code available is numpy. One of the problems in Python is that the object-oriented nature of the language is the source of its inefficiencies. With no strict types, Python always needs to check parameters on every operation. Numpy provides a new datatype, the array, which helps solve some of these issues. Arrays can only hold one type of object, and because Python knows this it can use some optimisations to speed things up to almost what you can get from writing your code directly in C or FORTRAN. The classic example of the difference is the for loop. Let's say you wanted to scale a vector by some value, something like a*b. In regular Python, this would look like:

```
django-admin startproject newsite
```



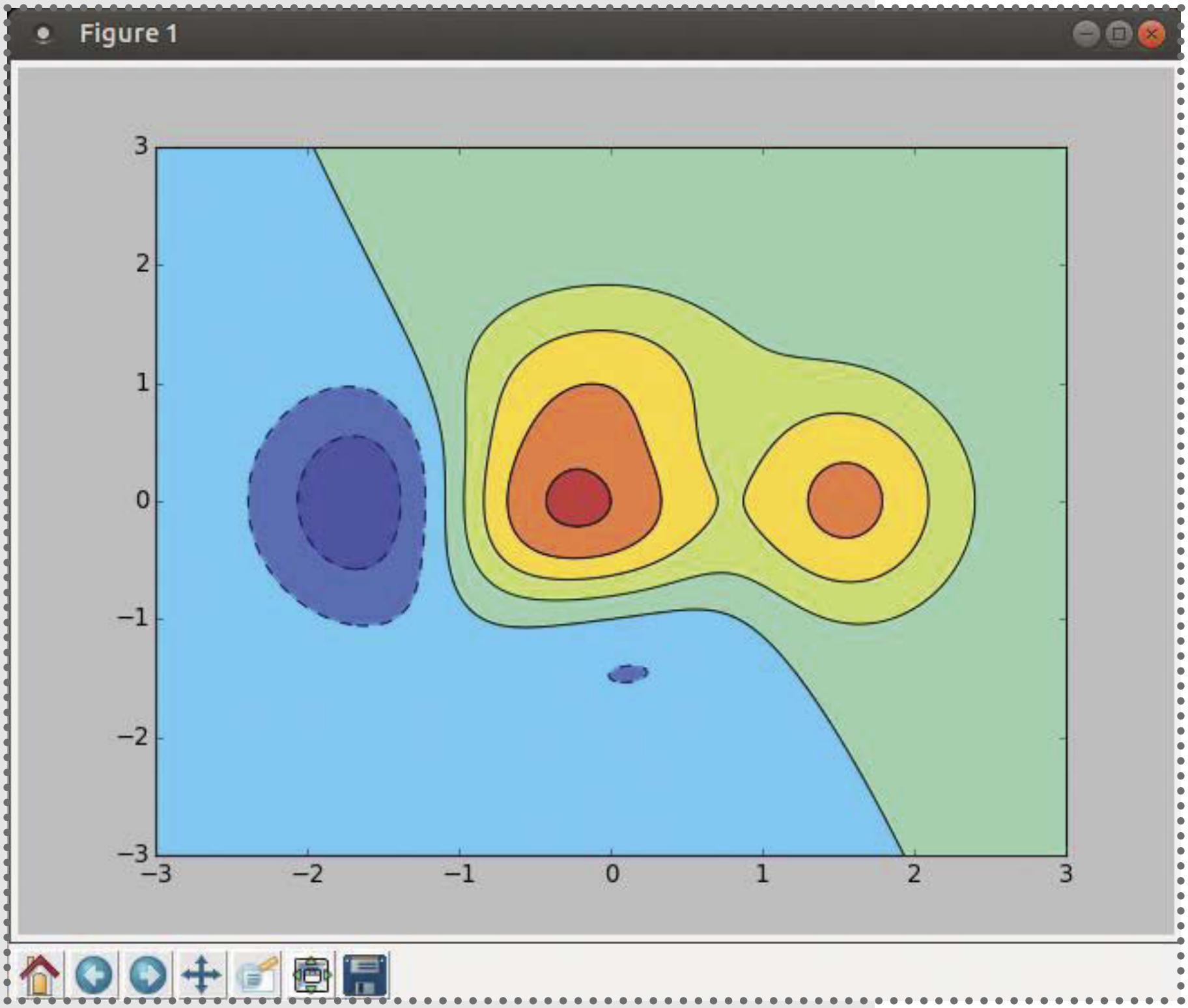

```
for elem in b:
 c.append(a * elem)
```

In numpy, this would look like:

```
a*b
```

So, not only is it faster, it is also written in a shorter, clearer form. Along with the new datatype, numpy provides overloaded forms of all of the operators that are of most use, like multiplication or division. It also provides optimised versions of several functions, like the trig functions, to take advantage of this new datatype. The largest package available, that is built on top of numpy, is scipy. Scipy

Above The numpy package makes it easy to visualise your data



provides sub-sections in several areas of science. Each of these sub-sections need to be imported individually after importing the main scipy package. For example, if you are doing work with differential equations, you can use the "integrate" section to solve them with code that looks like

```
import scipy
import scipy.integrate
result = scipy.integrate.quad(lambda x: sin(x),
0, 4.5)
```

Left The ability to generate complex plots is essential

Differential equations crop up in almost every scientific field. You can do statistical analysis with the "stats" section. If you want to do some signal processing, you can use the "signal" section and the "fftpack" section. This package is definitely the first stop for anyone wanting to do any scientific processing.

Once you have collected your data, you usually need to graph it, in order to get a visual impression of patterns within it. The primary package you can use for this is matplotlib. If you have ever used the graphics package in R before, the core design of matplotlib will be familiar as it has borrowed quite a few ideas. There are two categories of functions for graphing: low-level and high-level. High-level functions try to take care of as many of the menial tasks as possible, like creating a plot window, drawing axes, selecting a coordinate system, etc. The low-level functions give you control over almost every part of a plot, from drawing individual pixels to controlling every aspect of the plot window. It also borrowed the idea of drawing graphs into a memory-based window. This means it can draw graphs while running on a cluster.

If you need to do symbolic maths, you may be more used to using something more like Mathematica or Maple. Luckily, you have sympy, which can be used to do many of the same things. You can use Python to do symbolic calculus, or to solve algebraic equations. The one weird part of sympy is that you need to use the symbols() function to tell sympy what variables are valid to be considered in your equations. You can then start doing manipulations using these registered variables.

You may have large amounts of data that you need

The Need for Speed

Sometimes you need as much speed as you are capable of pushing on your hardware. In these cases, you always have the option of using Cython. This lets you take C code from some other project, which has probably already been optimised, and use it within your own Python program. In scientific programming, you are likely to have access to code that has been worked on for decades and is highly specialised. There is no need to redo the development effort that has gone into it.

to work with and analyse. If so, you can use the pandas package to help deal with that. Pandas has support for several different file formats, like CSV files, Excel spreadsheets or HDF5. You can merge and join datasets, or do slicing or subsetting. In order to get the best performance out of the code, the heaviest lifting is done by Cython code that incorporates functions written in C. Quite a few ideas on how to manipulate your data were borrowed from how things are done in R.

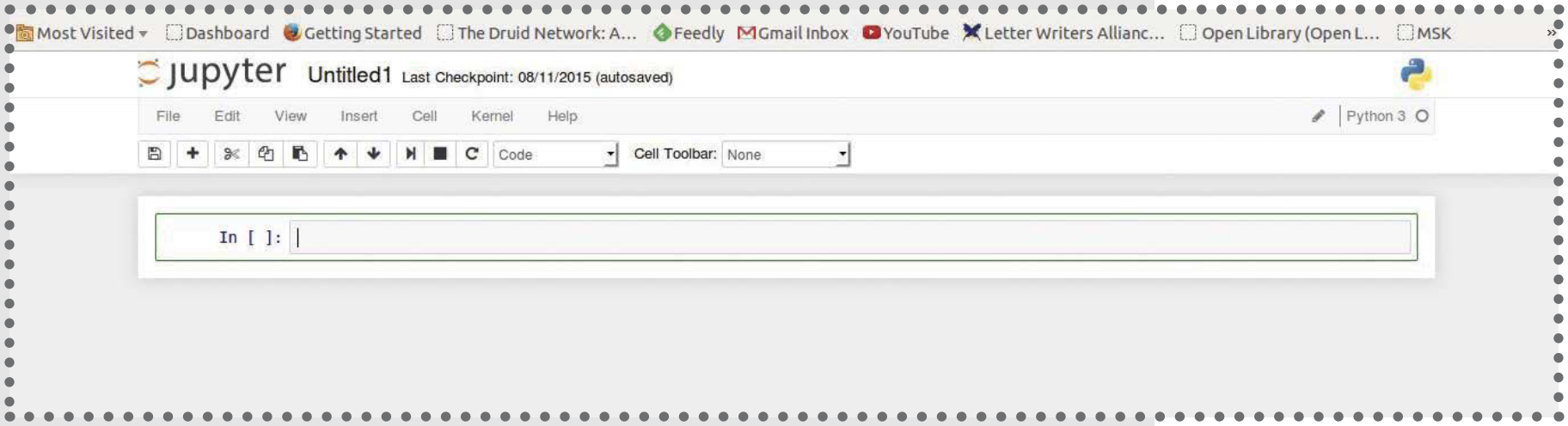
You now have no reason not to start using Python for your scientific work. You should be able to use it for almost any problem that comes up!

"Python knows what it isn't good at; it can incorporate code from C or FORTRAN"

Interactive science with jupyter

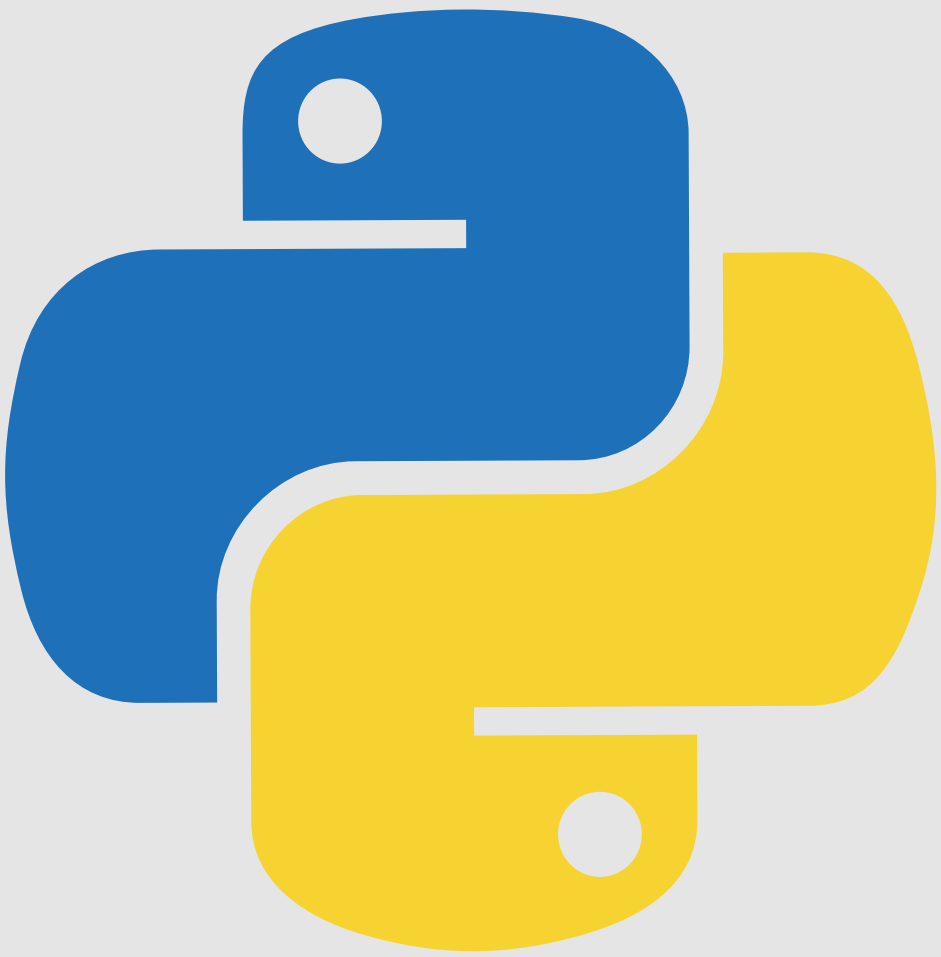
For a lot of scientific problems, you need to play with your data in an interactive way. The original way you would do this was to use the IPython web notebook. This project has since been renamed Jupyter. For those who have used a program like Mathematica or Maple, the interface should seem very familiar. Jupyter starts a server process, by default on port 8888, and then will open a web browser where you can open a worksheet. Like most other programs of this type, the entries run in chronological order, not in the order that they happen on the worksheet. This can be a bit confusing at first, but it means that if you go to edit an earlier entry, all of the following entries need to be re-executed manually in order to propagate that change through the rest of the computations.

Jupyter will correctly print mathematical expressions within the produced web page, as it supports the appropriate formatting. You can also mix documentation



blocks and code blocks within the same page. This means that you can use it to produce very powerful educational material, where students can read about the techniques, and then actually run it and see it in action. By default, Jupyter will also embed matplotlib plots within the same worksheet as a results section, so you can see a graph of some data along with the code that generated it. This is huge in the growing need for reproducible science. You can always go back and see how any analysis was done and reproduce any result.

Above Jupyter Notebook is a web application for creating and sharing documents that contain live code and equations



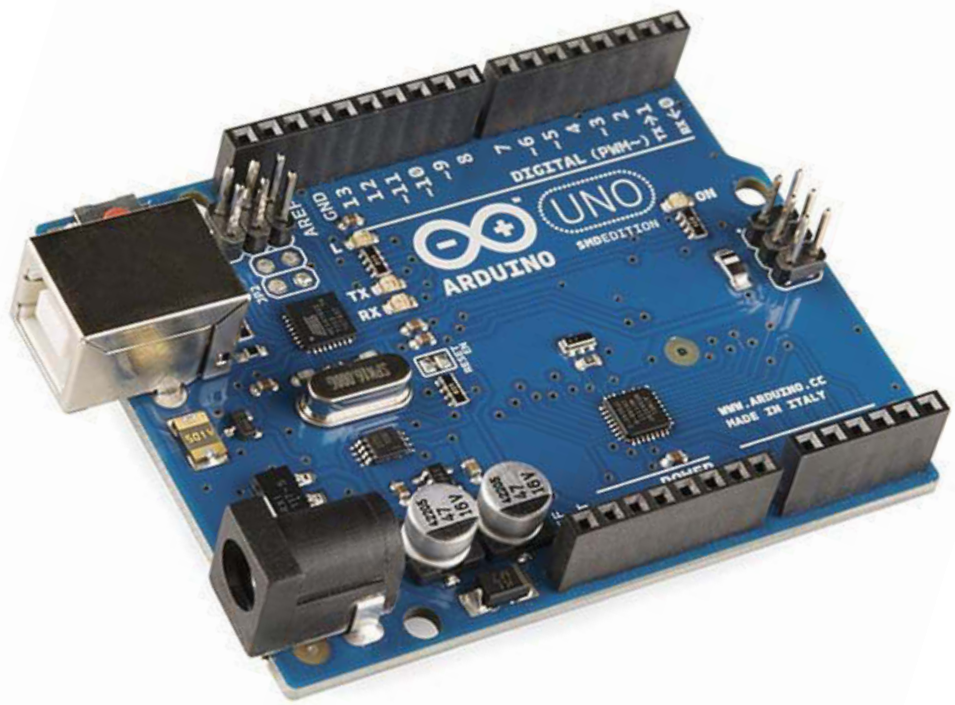
Robotics and electronics

See your code come to life in the real world around you with physical applications of robot technology

Robotics is the most direct way that your code can interact with the world. It can read actual sensor information and move real actuators and get real work done.

The first thing your robot needs is the ability to sense the world around it. The one sense that we as humans feel is most useful is sight. With web cameras being so cheap and easy to connect to hardware, vision is easy to give to your robot. The real problem is how to interpret this data. Luckily, you can use the OpenCV project to do just that. It is a vision package that can provide simple image gathering and processing, to extremely complex functions like face recognition and extraction of 3D objects. You can identify and track objects moving through your field of view. You can also use OpenCV to give you robot some reasoning capabilities, too. OpenCV includes a set of functions for machine learning, where you can do statistical classification or data clustering, and use it to feed decision trees or even neural networks.

Another important sense that you may want to use is sound. The Jasper project is one that is developing a complete voice control system. This project would give you the structure you need to give your robot the ability to listen for and respond to your verbal commands. The project has



gotten to the point where you can give it a command and the voice recognition software can translate this into text. You then need to build a mapping of which pieces of text correspond to which commands to execute.

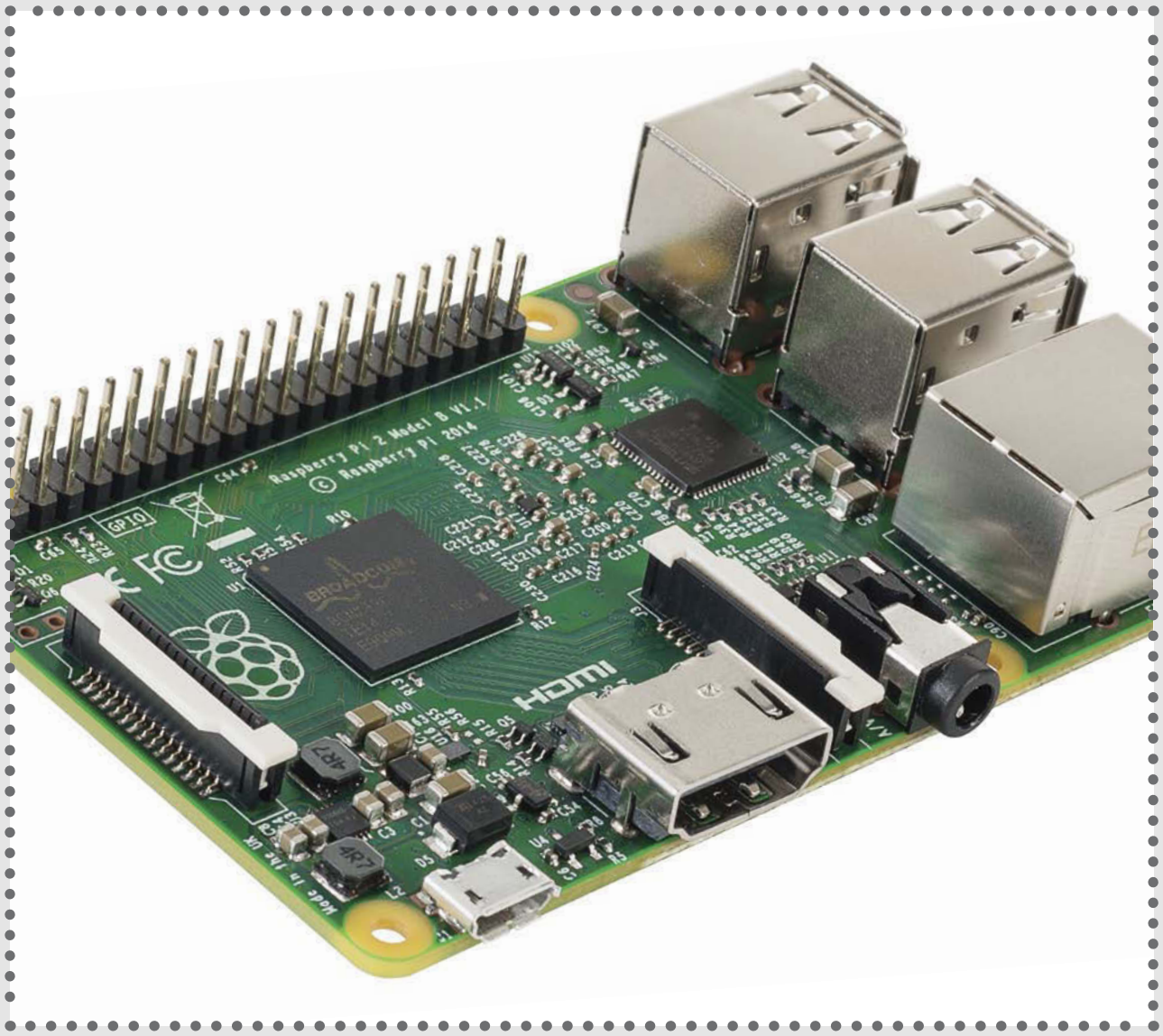
There are lots of other sensors you can use, but this begins to leave the realm of store-bought hardware. Most other sensors, like temperature, pressure, orientation or location, need specialised hardware that needs to be interfaced to the computer brain for your robot. This means it is time to get your soldering iron out. As for reading the data in, this is most often done over a basic serial connection. You can then use the pySerial module to connect to the serial port and read data off the connection. You can use:

```
import serial
```

... to load the module and start talking to your sensor. The problem is that this is a very low-level way to communicate. You, as the programmer, are responsible for all of the details. This includes communication speed, byte

Arduino

In contrast to the Raspberry Pi, which runs a full OS from its SD card, the Arduino boards are microcontrollers rather than full, complete computers. Instead of running an OS, the Arduino platform executes code that is interpreted by its firmware. It is mainly used to interface with hardware such as motors, servos, sensors, etc.



size, flow control; basically everything. So this will definitely be an area of your code where you should plan on spending some debugging time.

Now that you have all of this data coming in, what will you do with it? You need to be able to move actuators out in the world and have real effects. This could be motors for wheels or tracks, levers to shift objects, or potentially complete limbs, like arms or legs. While you could try and drive these types of electronic devices directly from the output ports of your computer, there usually isn't enough current available to provide the necessary power. So, you will need to have some off-board brains capable of handling the supplying of power to these devices.

One of the most popular candidates for this task is the Arduino. Luckily, the Arduino is designed to connect to the serial port of your computer, so you can simply use pySerial

Raspberry Pi

While we haven't discussed what kind of computer to use for your robotics project, you should consider of course the Raspberry Pi. This tiny computer should be small enough to fit into almost any robot structure that you might be building. Since it is already running Linux and Python, you should be able to simply copy your code development work to the Pi. It also includes its own IO bus so that you can have it read its own sensors.

to talk to it. You can send commands to code that you have written and uploaded to the Arduino to handle the actual manipulations of the various actuators. The Arduino can talk back, however. This means that you can read feedback data to see what effect your movements have had. Did you end up turning your wheels as far as you wanted to? This means that you could also use the Arduino as an interface between your sensors and the computer, thus simplifying your Python code even more. There are loads of add-on modules available, too, which might be able to provide the sensing capabilities you need right out of the box. There are also several models of Arduino, so you may be able to find a specialised one that best fits your requirements.

Now that you have all of this data coming in and the ability to act it out in the real world, the last step is giving your robot some brains. This is where the state of the art does not live up to the fantasy of R2-D2 or C-3PO. Most of your actual innovative coding work will likely take place in this section of the robot. The general term for this is artificial intelligence. There are several projects currently underway that you could use as a starting point to giving your robot some real reasoning capability, like SimpleAI or PyBrain.

"Sensors like temperature need specialised hardware"

ROS – Robot Operating System

While you could simply write some code that runs on a standard computer and a standard Linux distribution, this is usually not optimal when trying to handle all of the data processing that a robot needs when dealing with events in real-time. When you reach this point, you may need to look at a dedicated operating system – the Robot Operating System (ROS). ROS is designed to provide the same type of interface between running code the computer hardware it is running on, with the lowest possible overhead. One

of the really powerful features of ROS is that it is designed to facilitate communication between different processes running on the computer, or potentially over multiple computers connected over some type of network. Instead of each process being a silo that is protected from all other process, ROS is more of a graph of processes with messages being passed between them.

Because ROS is a complete operating system, rather than a library, it is wrong to think that you can use it in your Python code. It is better to think that you can write Python code that can be used in ROS. The fundamental idea is to be as agnostic as possible; interfaces to your code should be clean and not particularly care where they are running or who is talking to them. Then, it can be used within the graph of processes running within ROS. There are standard libraries available that allow you to do coordinate transformations, useful for figuring out where sensors or limbs are in space. There is a library available for creating preemptible tasks for data processing, and another for creating and managing the types of messages that can be handed around the various processes. For extremely time-sensitive tasks, there is a plugin library that allows you to write a C++ plugin that can be loaded within ROS packages.

Bypassing the GIL

For robotics work, you may need to run some code truly in parallel, on multiple CPUs. Python currently has the GIL (Global Interpreter Lock), which means that there is a fundamental bottleneck built into the interpreter. One way around this is to actually run multiple Python interpreters, one for each thread of execution. The other option is to move from CPython to either Jython or IronPython, as neither has a GIL.

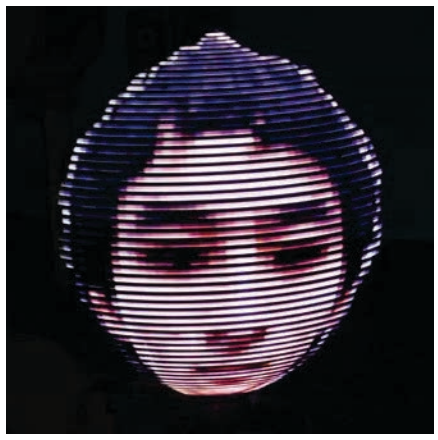
Pixel Globe

Evan Kahn and Kyle Lund harness the persistence of vision phenomenon to create a futuristic hologram-like display

What is the persistence of vision phenomenon?

Evan Kahn: When something moves, your eye can't immediately update the things that it sees, so if something's moving really fast then you see a kind of blur, you see something in multiple positions at the same time. You can take advantage of that by just moving something really fast and then changing the way that it looks at every time interval.

That's sort of how a CRT display works. This is the same except it's mechanically moving fast, and we update the lights – it's a plastic ring with LEDs around the outside, and it rotates at 500 rpm, which is 8.3 times per second, which means that if we want 150 pixels around the sphere, which is what we found looks good, we need to change what the lights are displaying once every 0.6 ms.



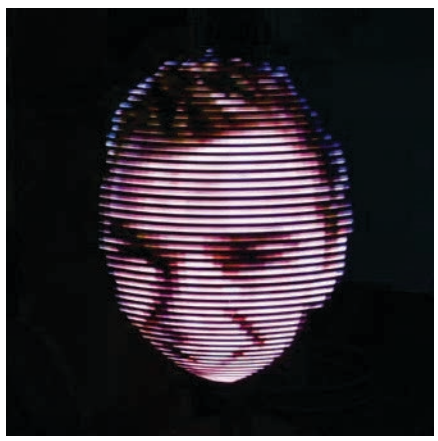
Evan Kahn is an engineering student at Harvey Mudd College interested in music, activism, and human-computer interaction. Find him at **http://eka.hn**

How does the Pixel Globe work?

Kyle Lund: We have one section of the project that's dedicated to just spinning the motor and the LED setup, so that's run with an FPGA, and that's running just a control loop to keep it at a really stable speed. Then we have the Raspberry Pi, which is actually controlling the LEDs, so it just assumes that the ring is spinning at 500 rpm and updates pixel values to the whole array, so you get that stationary image all the way around.

How does the FPGA system work?

KL: The motor control module is all written in SystemVerilog on an FPGA. We have a digital encoder on the motor, and so that outputs a square wave. The



Kyle Lund is a robotics major at Harvey Mudd College, class of 2017. He's from Boulder, Colorado, and spends his free time running, skiing and playing piano

frequency of that square wave is related to the speed of the motor, so I read that in on the FPGA and I get a period reading, and then I pump that through a PI [proportional-integral] control loop that's set at the target speed of 500 rpm.

Then that gets put out through another module that converts an input to a PWM signal, and then that PWM signal is sent back to the motor to keep it running at 500 rpm. The FPGA is a Field Programmable Gate Array – it's a programmable digital logic board. You can put AND gates, OR gates, that kind of logic onto that board, so everything on that board is built from basically the logic gate level, with a little bit of extra abstraction.

EK: That's how SystemVerilog works – you write what looks like C and then it basically turns your C into LEGO building blocks of logic gates, which it arranges into the correct patterns. It breaks your C down into logic-level arguments that can be converted into actual hardware, and then in the FPGA chip itself the logic gates are rewired to plug into each other. You can get arbitrarily complicated – Kyle used it to build a motor control. People prototype computer processors on these things.

What exactly is the Pi handling?

EK: I pass in a .led file to this program, and the .led file is basically a 3D array – you have a 3D array of pixel colour values in rows of columns, like a grid, and the grid is the pixels that you see on the globe in this 2D array, and then you have multiple of those to make a 3D array – and that's how you can make animations.

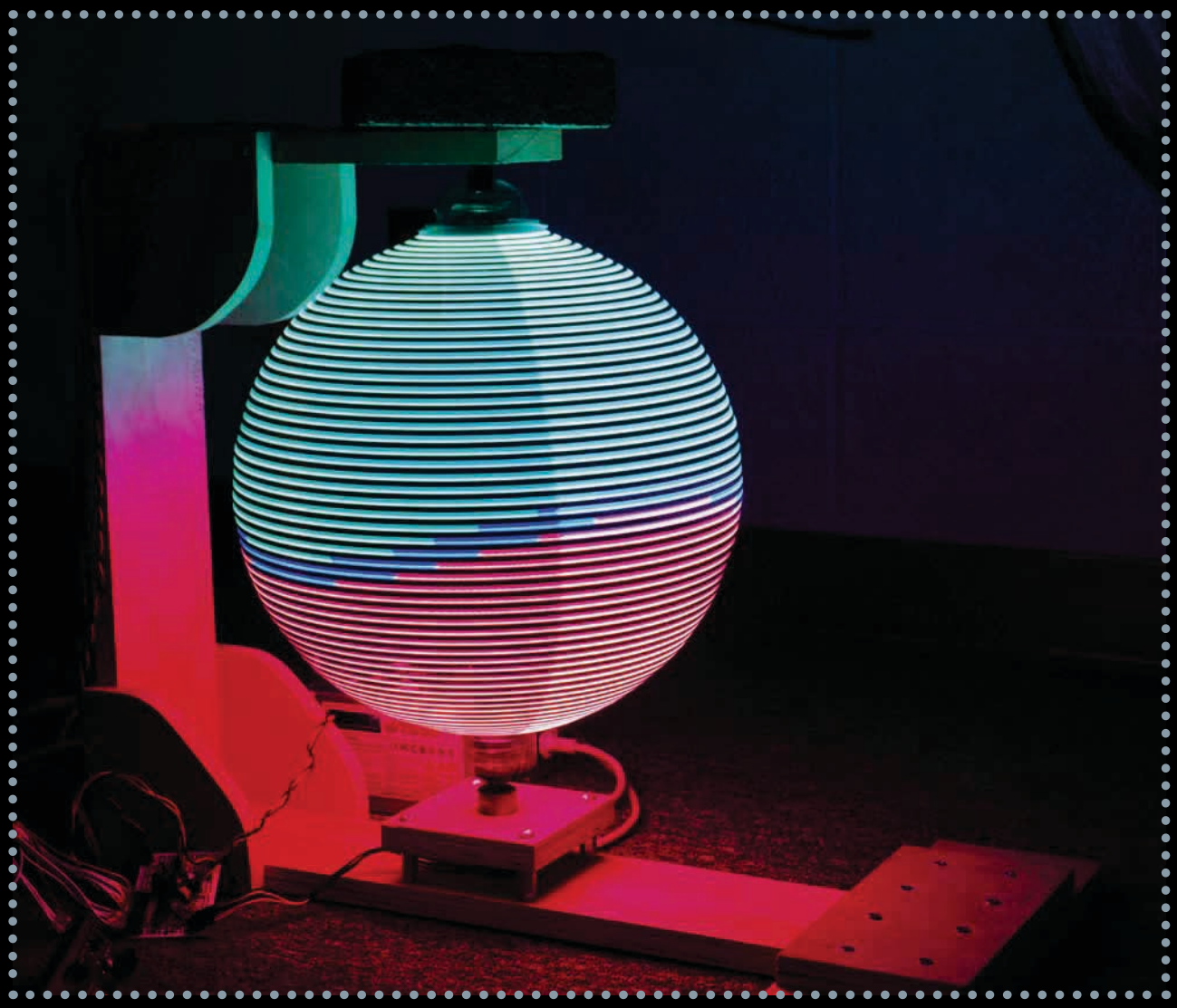
THE PROJECT ESSENTIALS

- Raspberry Pi 2 Model B
- µMudd FPGA board
- Adafruit Perma-Proto 1/2-sized breadboard
- Adafruit 0.5m DotStar LED strip
- Pololu 9:7:1 Gearmotor
- Miniature slip ring
- D44VH10 power transistor
- Salvaged computer PSU
- 74AHCT125 Quad level-shifter

What the Pi does is it reads that into memory. The LEDs are one long strip, right? And the strip is the vertical column of the globe. So the Raspberry Pi writes out to each vertical column every 0.6 ms, it changes the colour values every time. And if you have an animation keyed in, every time the globe completes one revolution it goes to the next frame and then it writes the next frame of animation.

How do you create the images that are displayed on the globe?

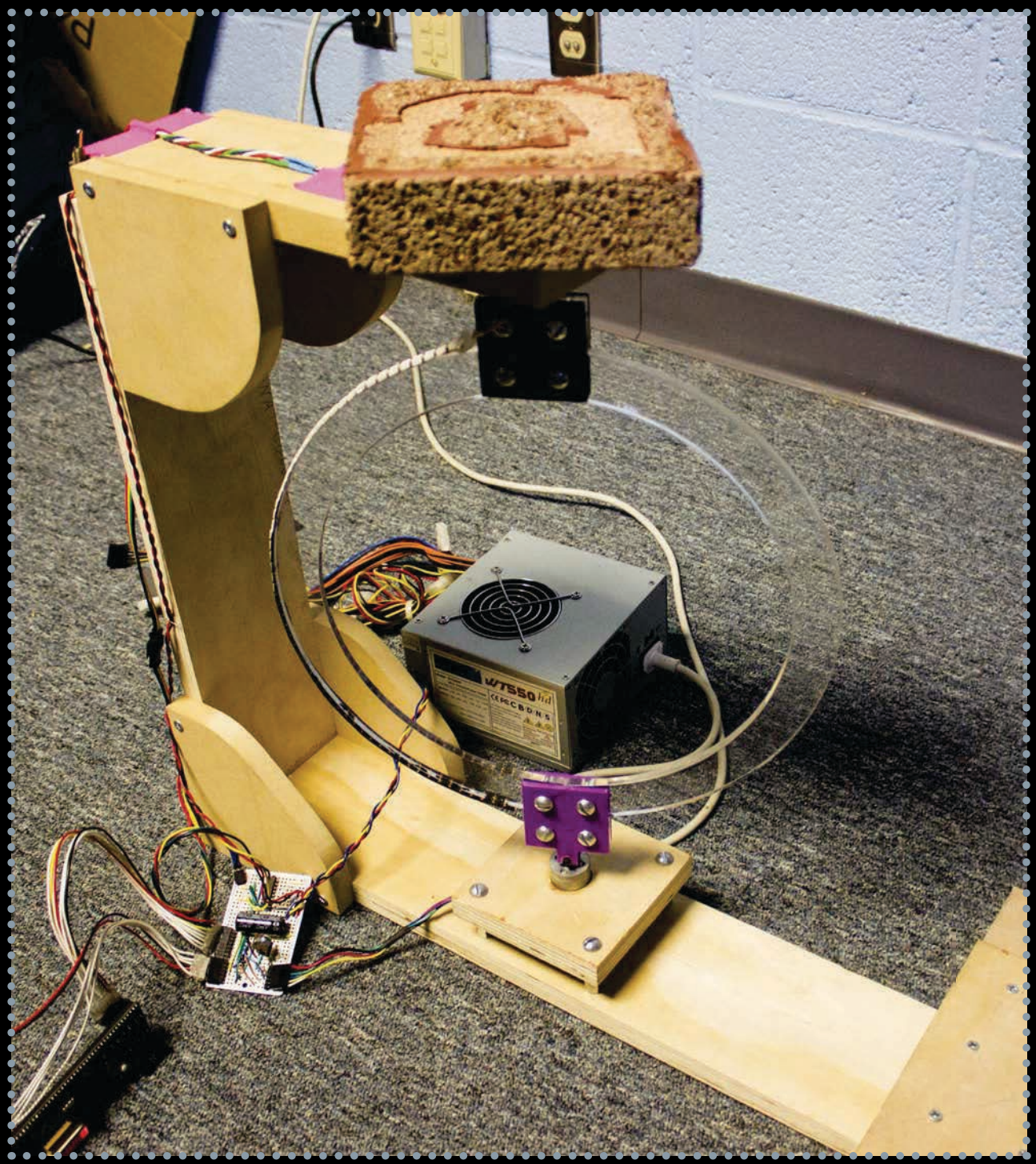
EK: There are two ways to do it: the first way is with the ncurses application, the second way is with a Python script we hacked together during class the day that we were supposed to present this. In hindsight, the second way would have been better to start with, but the ncurses application was fun.



Left It's difficult to capture video of this, but the sine wave photographed here is actually animated to oscillate along the circumference of the Pixel Globe

"It's a plastic ring with LEDs around the outside, and it rotates at 500 rpm, 8.3 times per second"

You tell it how big your image is going to be; the images all have to be 60 pixels tall, because that's how many LEDs are physically in the strip, but you can set it to as many wide as you want – we haven't really tested how wide your images can be because we found a value that, when you run it at 500 rpm, makes the pixels look roughly like they're the same width and height. When you type in the size of the structure that you want, it encodes that into the header of the file and then allocates memory for that, and when you read the .led file, it grabs the header and it figures out how much memory it needs to



Left Turning the house lights on dispels the sci-fi illusion and reveals a perfectly ordinary motor-driven LED strip, curved in a perfect arc

allocate for the frame data itself.

The second method is the same file format but… I mean, you can write only 20 lines of Python code – and this is insane to me, because it took us the week over Thanksgiving break writing C to do this, and then we wrote 20 lines of Python code that did the same thing. You can put together a .png file and the script will write the header for you, and then figure out the frame width and frame height, write all the pixel values – it handles everything all automatically. For editing things by hand and also doing animations, though, the ncurses application still works best.

Like it?

Interested in building your own Pixel Globe? Evan and Kyle use Adafruit's DotStar LEDs: **http://bit.ly/1RkdeEm.** Their project code can all be found on their GitHub page: **http://bit.ly/1P7ZEgL**. As for the FPGA, check out the MuddPi Mark IV Board schematics at **http://bit.ly/1KbABrY**

Further reading

There are some great build notes for this project over at Hackaday: **http://bit.ly/1Ogj9Xn**

Control lights with your Pi

The winter nights are getting longer; use Raspberry Pi and a mobile device to remotely control your lights

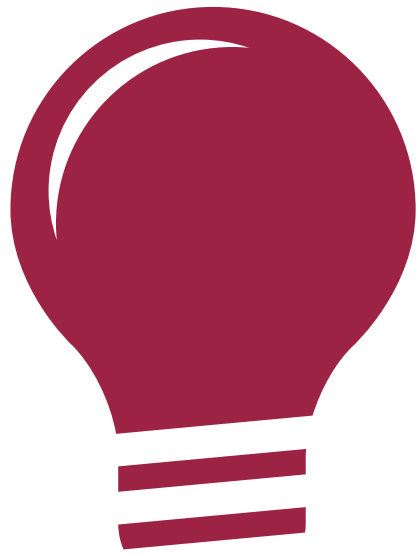
The folks at Energenie have created some genius plug sockets that can be turned on and off via your Raspberry Pi. You can buy a starter kit which includes the RF transmitter add-on board and two sockets to get you started. The add-on board connects directly to the GPIO pins and is controlled with a Python library. Once everything is installed and set up, your Raspberry Pi can be used with the Pi-mote to control up to four Energenie sockets using a simple program. This tutorial covers how to set up the software, the sockets and how to adapt the program to run on your mobile device.

THE PROJECT ESSENTIALS

Pi-Mote IR control board with RC sockets
http://bit.ly/1MdpFOU

Desk lamp

Accessories



01 Set up

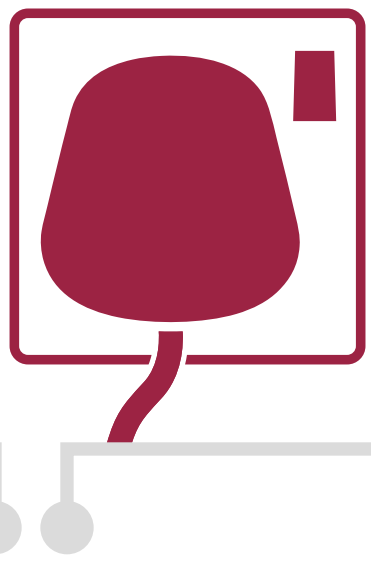
To get started, boot up your Raspberry Pi and load the LX Terminal, then update your software by typing:

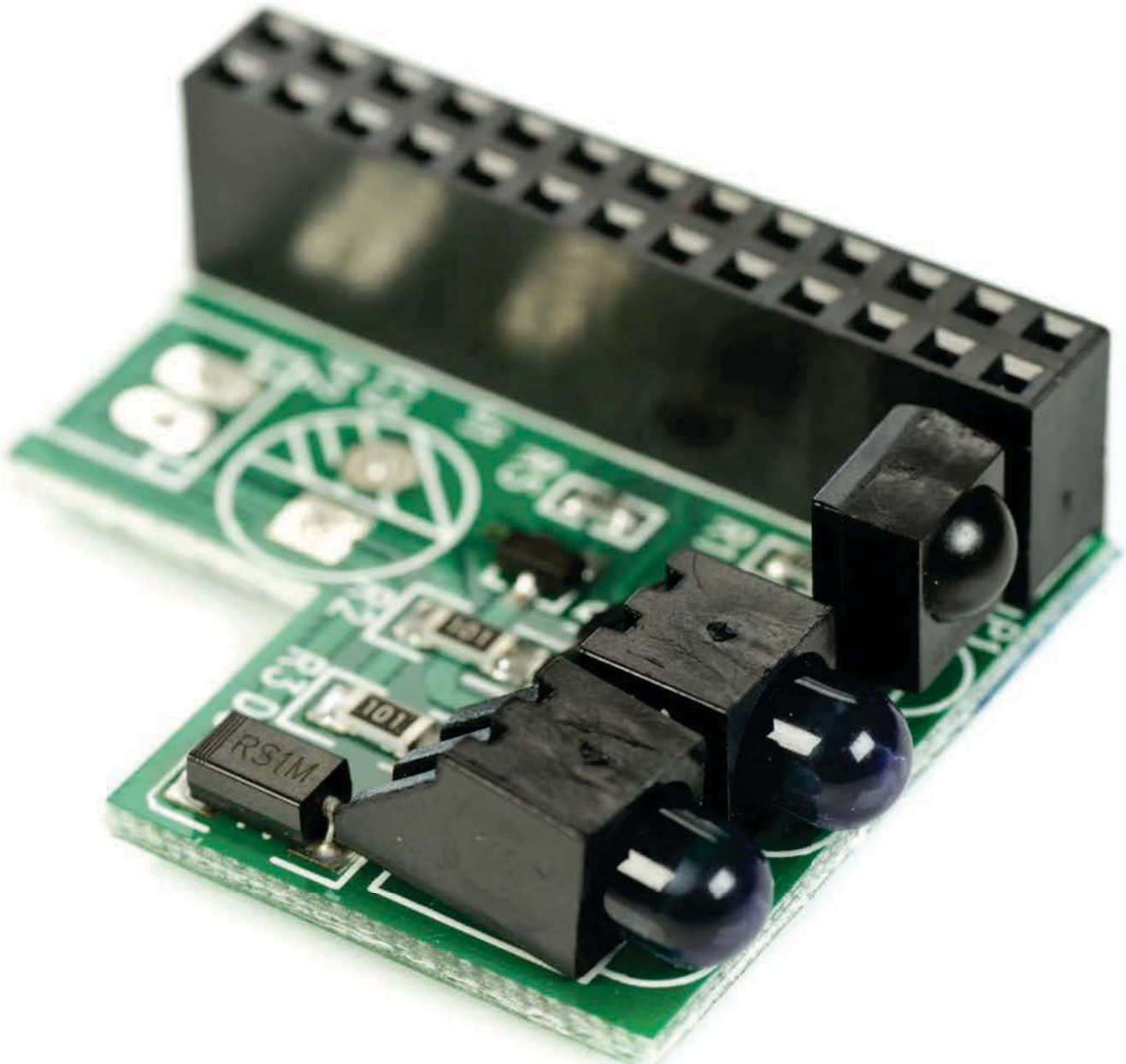
```
sudo apt-get update
sudo apt-get upgrade
```

Depending on which version of the OS you're using, you may need to install the Python GPIO libraries. (Raspbian Jessie comes with this library pre-installed, so you can skip this step.) Type:

```
sudo apt-get install python-rpi.gpio
```

On completion, reboot your Pi. This will install the Python GPIO libraries, meaning you can access and control the pins with Python code.





02 Install the Energenie library

Next, install the Energenie libraries. These enable the Pi-mote board and Raspberry Pi to interact with Python. In the LX Terminal, depending on which version of Python you are using, type either:

```
sudo apt-get install python3-pip
sudo pip-3.2 install energenie
```

…for Python 3, or:

```
sudo apt-get install python-pip
sudo pip install energenie
```

Above The Pi-mote transmitter is so easy to use; it is powered by the Pi and uses a transmit-only open loop system

...for an older version. In the future, Energenie will update its software and you may need to run a check for updates to ensure that you have the most recent version. To update the software, type the code:

```
sudo pip install energenie -update
```

03 Fitting the Pi-mote

Before fitting the Pi-mote transmitter, shut down your Raspberry Pi with sudo poweroff. Unplug the power supply and fit the module onto your Raspberry Pi. The 'L' part of the board fits opposite the HDMI port. Power up the Pi and plug in one of your Energenie sockets in the same room or area that your Pi is in. The range is fairly good, but furniture and walls may sometimes block the transmission signal. You can test that the socket is working by plugging in something like a desk lamp and then pressing the green button that is located on the socket. This will trigger the socket on and off, turning the lamp on and off.

04 Download the set-up code

Before the Raspberry Pi can interact with the socket and switch it on/off, it requires programming to learn a control code that is sent from the transmitter. Each socket has its own unique code so that you can control up to four individually. Energenie provides the set-up program which can be found inside your tutorial resources, so check those out for more.

05 Set up your socket

Once you have downloaded the set-up program, run it. This should place the socket into 'learning mode', and

IP address

Every device on the Internet is assigned an Internet Protocol address (IP address). This is a numerical label which is used to locate and identify each device within a network which may contain many thousands of devices. Most home network IP addresses start with the numbers 192.168, with your router being on 192.168.1.1.

will be indicated by the LED on the front of the socket housing slowly flashing. If it is not doing this, press and hold the green button for at least five seconds and release it when the LED starts to flash at one-second intervals. Run the program and it will send a signal out. Follow the on-screen prompts, pressing the return key when required.

When the code is accepted, success will be indicated with a brief flashing of the LED on the housing. If you have more than one socket to set up, simply use the same program and method to do so for as many times as required.

06 A quick test

Before you get to the task of creating your Python code to control your socket, it is always advisable to test that the socket is working correctly. Ensuring that the power is turned on at the wall plug and that the lamp is switched on, you can turn the lamp off by pressing the green button on the front of the Energenie socket. The lamp should turn back on again when the button is next pressed.

07 Code to turn the socket on

The Python Energenie library makes it incredibly easy to create a code to turn the socket on, which will then turn your lamp on. Before you know it, you will be using your Raspberry Pi to turn the kettle or the TV on or off!

Open your Python editor and start a new program. Next, import the Raspberry Pi GPIO library (line 1, below), then import the Energenie library (lines 2 and 3). Finally, add in the code to switch the socket on (line 4). Save and

then run your program. The socket will turn on, you may hear a click, and then your lamp will come on.

```
import RPi.GPIO as GPIO
import energenie
from energenie import switch_on
energenie.switch_on(1)
```

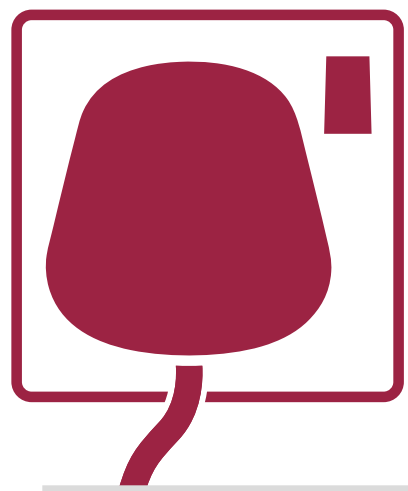
08 Switching the socket on and off

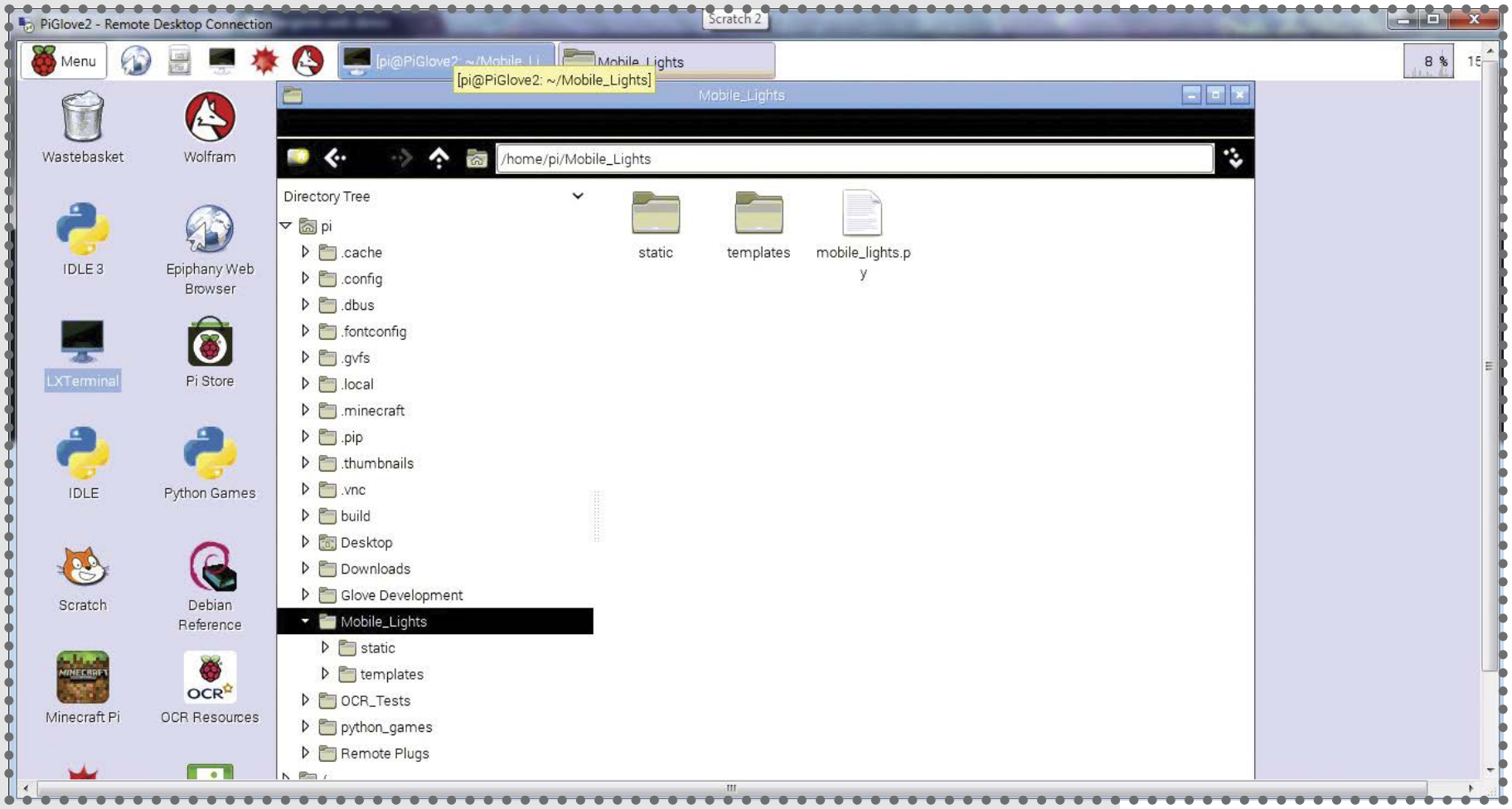
Since you have not told the socket to turn off, it will stay on, which means the lamp will stay on forever (or until the bulb blows)! To turn the socket off after five seconds, import the time function at the start of your program (line 2, below), add the command to turn off the socket (line 5). Then add a pause with the sleep command (line 7) and finally turn off the lamp (line 8). Now save and run the program.

```
import RPi.GPIO as GPIO
import time
import energenie
from energenie import switch_on
from energenie import switch_off

energenie.switch_on(1)
time.sleep(5)
energenie.switch_off(1)
```

09 Creating a web-based application

It is possible to augment this hack so that you can turn the lamp on and off from a mobile device such as your phone, laptop or tablet. This makes the whole project more impressive, slick and fun. The first step is to set up





your Raspberry Pi as a web server which will host and display a web page with the ON / OFF option. These buttons are interactive and control the socket. Open the LX Terminal and install pip and Flask:

```
sudo apt-get install pip
sudo pip install flask
```

10 CSS and HTML

To make the web page look presentable, you need to set up an HTML and a CSS file. HTML stands for HyperText Markup Language and is the markup language used to create web pages. Your browser reads HTML files and converts them into web pages, enabling images and objects to be embedded into the pages. Cascading Style Sheets, or CSS, is the code which describes how the web page will look; the presentation of the HTML content. It contains the instructions on how the elements will be rendered on your device. In this tutorial, it controls how

Above You'll need to get the folder names correct so that files are saved properly

the on and off options will be presented and look on the screen.

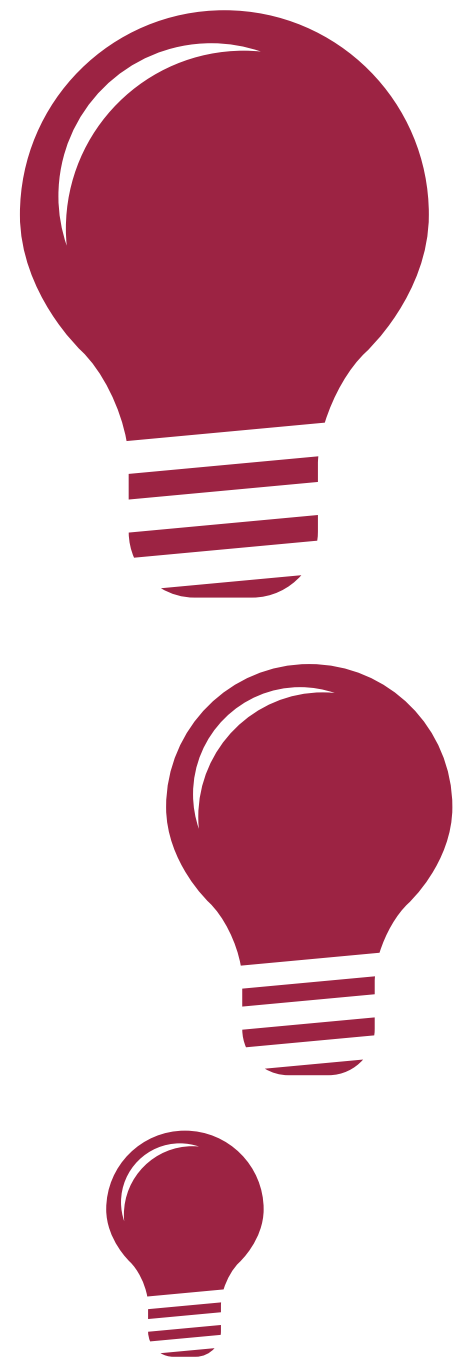
11 Create a new folder

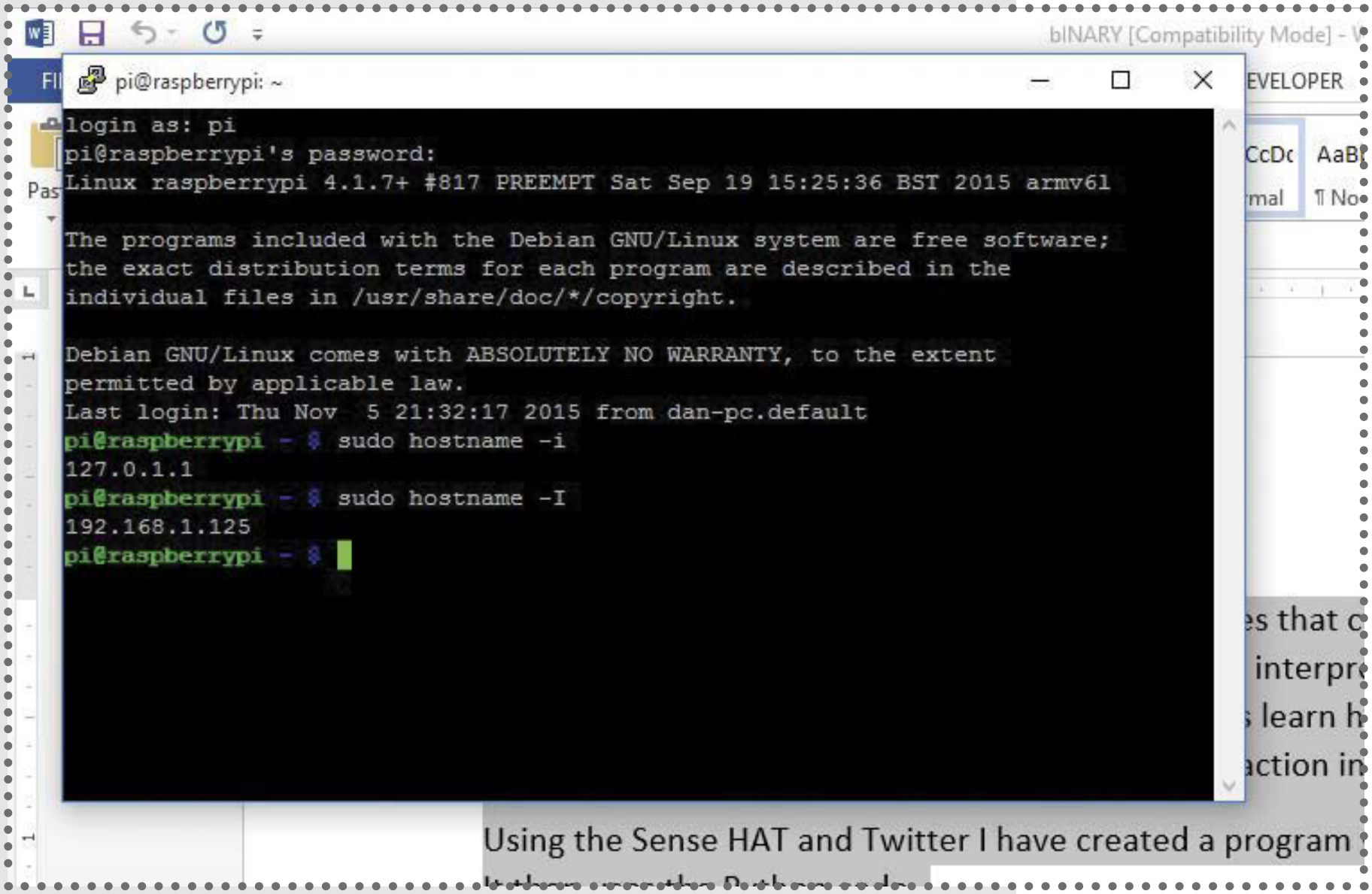
With Flask installed, reboot your Raspberry Pi; type sudo reboot. Create a new folder called Mobile_Lights in the /home/pi folder. This is where you will save the Python program which controls the socket and lamp, the CSS and the HTML file. You can create the folder in the LX Terminal by typing mkdir Mobile_Lights or right-clicking in the window and selecting New Folder.

12 The HTML files

Open the Mobile_Lights folder and create a new folder called 'templates'. This folder is where the HTML file is saved that contains the structure for the website layout. The code names the web page tab and, most importantly, adds the links for the on and off option. Open a text editor from your Start menu, or use nano and create a new file. Add the HTML below to the file and then save the file into the template folder, naming it 'index.HTML'. Remember, this is an HTML file and must end with the file extension .html:

```
<!doctype HTML>
<HTML>
<head>
<title>Light Controller</title>
<link rel="stylesheet" href="/static/style.
css" />
<meta name="viewport"
content="width=device-width, user-
scalable=no" />
```






```
</head>
<body>
<div class="on"><a href="/on/">ON</a></div>
<div class="off"><a href="/off/">OFF</a></
div>

</body>
</HTML>
```

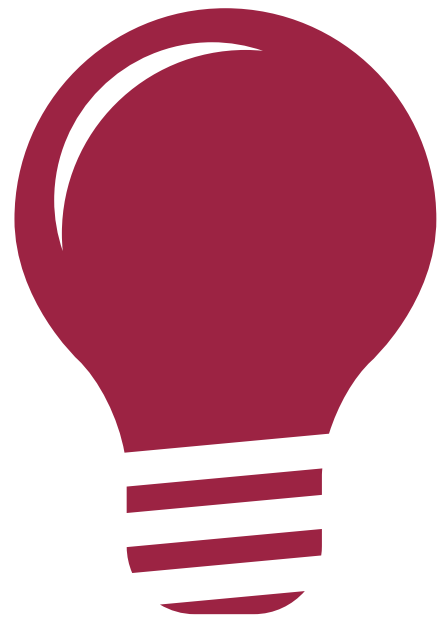
Above We recommend using the -I operator over -i with the hostname command, as the latter only works if the host name can be resolved

13 Add some style

The Cascading Style Sheet, CSS, is used to create and apply a 'button' style effect to the web page. Move back to the Mobile_Lights folder and create a new folder named 'static'. This is where the CSS file is saved. Create another new text file and add the code below, which sets out the 'style' for the web page. You can customise

the colours of the buttons from line 20 onwards. Save the file as 'style.css' in the static folder. Keep in mind that this is a CSS file and needs to be saved with the file extension .css:

```
body {
  position: absolute;
  margin: 0;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  font-family: Arial, sans-serif;
  font-size: 150px;
  text-align: center;
}
  width: 100%;
  height: 50%;
}

div a {
  width: 100%;
  height: 100%;
  display: block;
}

div.on {
  background: black;
}

div.on a {
  color: white;
```



```
}

div.off a {
  color: black;
}

a:link, a:visited {
  text-decoration: none;
}
div {
  display: block;
```

14 Putting it all together

The final part of the setup is to write the Python script that combines the index.html and style.css files with the Energenie socket control code similar to the one used in Step 7.

Open IDLE and start a new window, add the following code and save into your Mobile_Lights folder, naming it 'mobile_lights.py'. Line 4 uses the route() decorator to tell Flask the HTML template to use to create the web page. Lines 7 and 11 uses app.route('/on/') and app.route('/off/') to tell Flask the function to trigger when the URL is clicked. In line 15 the run() function is used to run the local server with our application. The if__name__ == '__main__': makes sure the web server only runs if the script is executed directly from the Python interpreter and not used as an imported module.

```
from flask import Flask, render_template
from energenie import switch_on, switch_off

app = Flask(__name__)
```

Flask

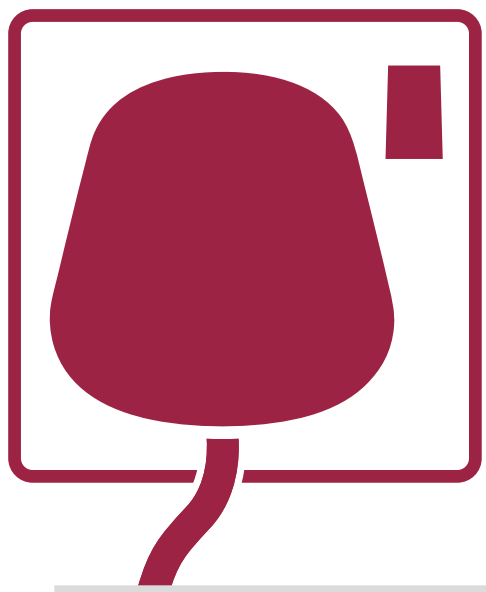
Flask is a powerful tool for creating interactive web pages and apps. If you are interested in learning more and trying out some other projects, this resource is a great place to start: **http://flask.pocoo.org.** Check out the site for examples of where Flask and Python are used for real-world applications and solutions: **http://flask.pocoo.org/community/poweredby.**

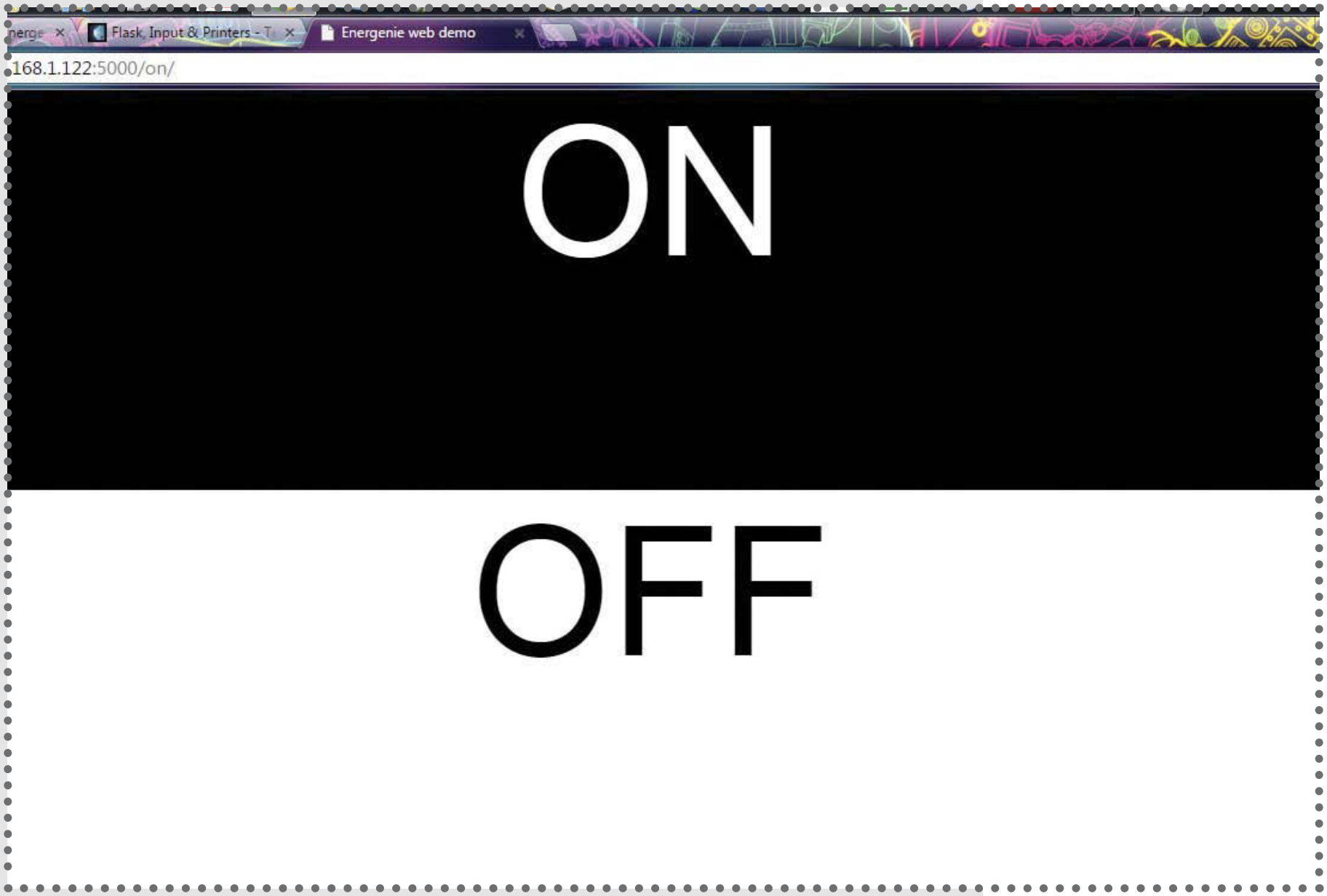
```
@app.route(‘/’)

def index():
     return render_template(‘index.HTML’)

@app.route(‘/on/’)
def on():
     switch_on()
     return render_template(‘index.HTML’)

@app.route(‘/off/’)
def off():
     switch_off()
     return render_template(‘index.HTML’)

if __name__ == ‘__main__’:
     app.run(debug=True, host=’0.0.0.0’)
```

15 Find your IP address

Before you start the web server running, you need to check the following:

- You have a folder called Mobile_Lights
- In the Mobile_Lights folder is a Python file named mobile_lights.py
- Also within the Mobile_Lights folder are two folders, one named templates which stores the index.HTML file and another folder named static which contains the file style.css





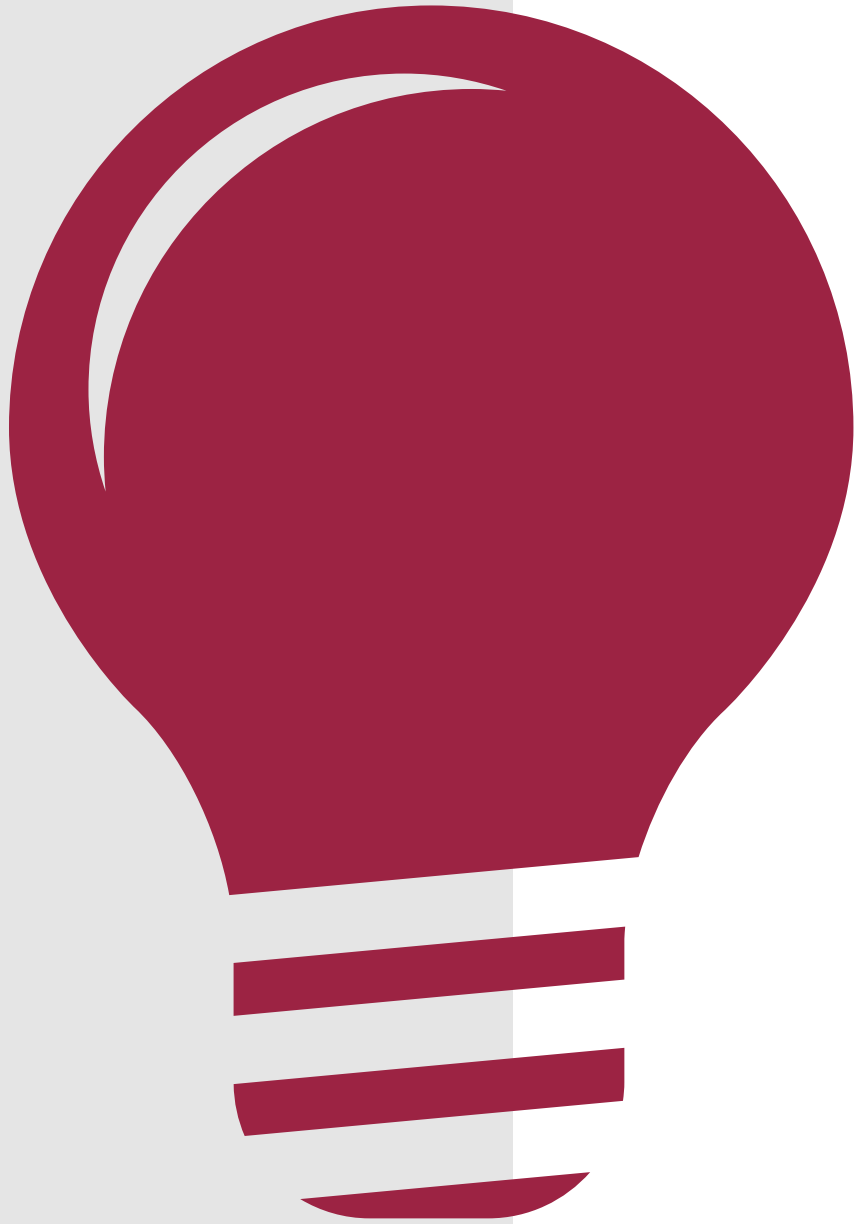
If all checks out, in the LX Terminal type sudo hostname –I. This will display the IP address of your Raspbery Pi – for example, 192.158.X.X. Make a note of it because this is the address you will enter into the web browser on your mobile device.

16 Start the web server

You have arrived at the point where you are ready to start the web server. Move to the Mobile_Lights folder by typing cd Mobile_Lights. Now run the Python mobile_lights.py program by typing sudo python mobile_lights.py. This starts up the web server, which is then ready to respond to the buttons that are pressed on the web page.

17 Turn your lights on and off

Grab your mobile device, smartphone or tablet and load the web browser. In the address bar enter the IP address that you noted down in Step 15. At the end of the address, add ':5000' – for example, 192.168.1.122:5000. The 5000 is the port number that is opened to enable the communication between your device and the Raspberry Pi. You will be presented with ON and OFF options, and you can now control the socket and whatever you have plugged in – kettle, radio, TV – all from your mobile device by simply pressing ON or OFF. Have fun!



Code a Tempest clone in FUZE BASIC Part 2

Remake a classic game in FUZE BASIC and delve into the world of programming

Welcome to part two of our FUZE BASIC tutorial. You will need to FUZE BASIC installed for this, and it can be downloaded for free for Linux and Raspberry Pi users from **www.fuze.co.uk/getfuzebasic**.

Don't worry; it's possible to skip part one and just jump straight in here. However, because this is a fairly large program, you won't be typing in off the page. Instead you will need to download the program listing from the Fuze website: **www.fuze.co.uk/tutorials/73MP357PART2.fuze**.

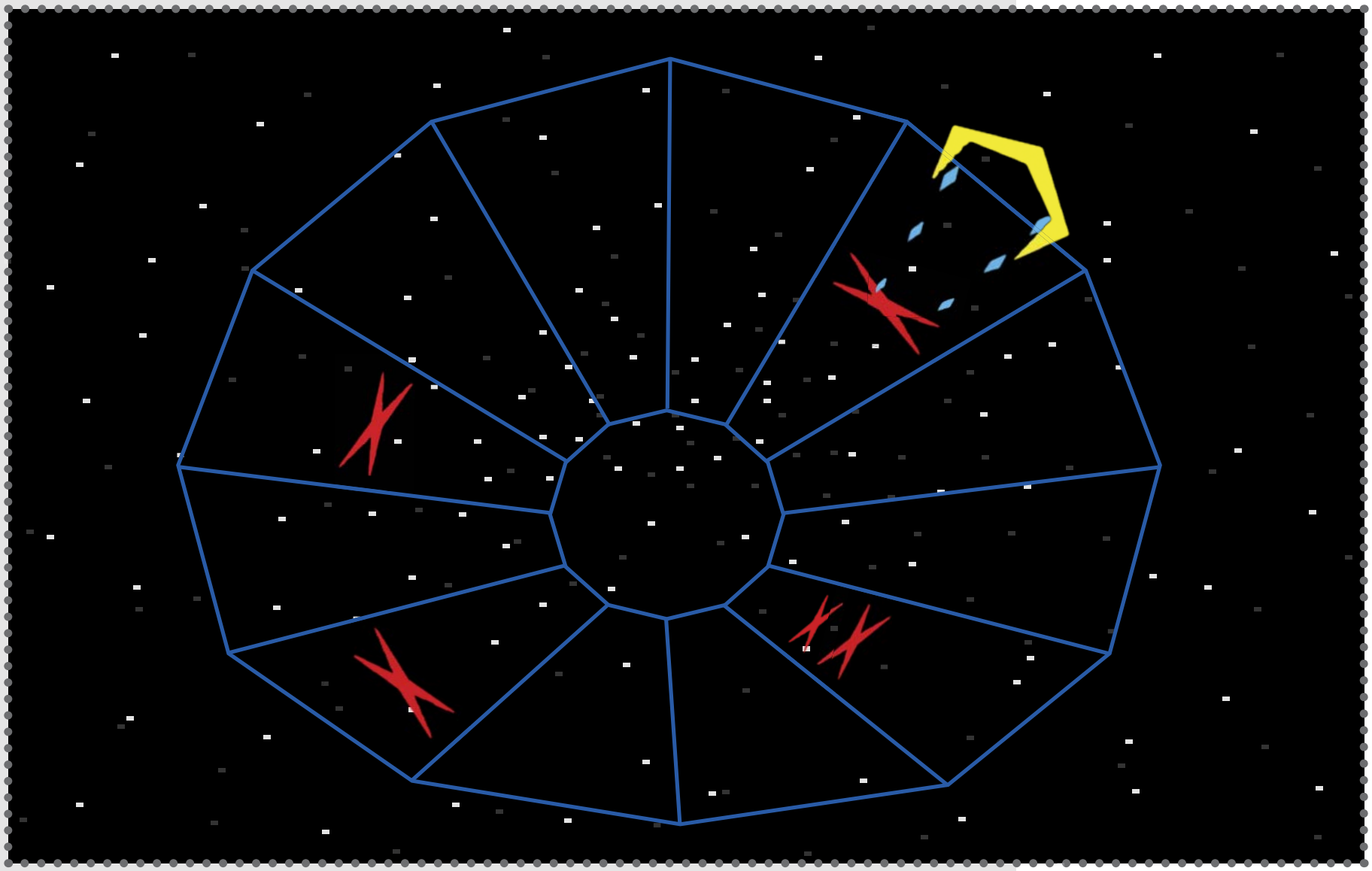
73MP357 is coded by Luke Mulcahy, FUZE's resident coder and, in the nicest way possible, our very own human supercomputer. This issue he tasks his brain with developing the basic game structure further, adding all sorts of awesome trinkets to delight. These include simple scoring, power-ups, enemies and the all-important fire power. Gameplay is also improved with smooth movement. In short, all you need for a rudimental – but definitely playable – game.

THE PROJECT ESSENTIALS

FUZE BASIC V3

https://www.fuze.co.uk/getfuzebasic

73MP357PART2.fuze



01 Grab the Part 2 code

Start FUZE BASIC and either load (with F8) 73MP357PART2.fuze or copy and paste the code straight into the editor (F2 switches between the editor and immediate mode). If you followed the first part last month then you'll be pleased to see things are coming along nicely. Run the program by pressing F3 or typing RUN in immediate mode.

02 Test it out

The first thing you'll notice is that we now have a score displayed. Tap the space bar and move left and right. We have fire power; hold on for a second longer and we should also have enemies and the all-important 'Power Up', which in this case awards us with a Particle Laser. We also have basic collision so enemies can be shot, but more importantly, if one touches you then you die too!

Have a play then bring up the editor; press Esc to stop the program then F2. We'll be brief on the sections we covered last month and offer more detail on the new additions.

03 Set up environment

Our initial section sets up a few environment variables. There are a number of variables used for screen resolution and frames per second (targetfps, minfps and maxfps). Luke has built in a very cool dynamic resolution feature that automatically adjusts the screen resolution to maintain a smooth frame rate. At the start of each game you can see the screen size adjust until it finds its optimum resolution to frame rate ratio. Once it reaches 60fps, a time delay is used to keep it there. We'll come back to this later but throughout the code you will find functions using the frame rate to ensure everything happens in sync. The remaining variables are straightforward.

04 Set up star field

The star field was covered last issue, but in brief this creates three variable arrays to store the positions, speed and angles of up to 1,000 stars. The arrays are populated with random initial settings.

05 Laser variables

The good stuff. Again a few arrays are set up to store the positional and movement information for each of the active lasers. Notice also laserReload = INT (targetfps / 2) on line 53, the first use of a frame synced action. In this case there is a counter so that the lasers can't fire too quickly but regardless of the frame rate, they will reload at the same frequency.

06 Enemy variables

There's just no point in having great firepower if we can't use it to exterminate a relentless supply of alien monsters hell-bent on taking over the Earth. You should be noticing something of a pattern by now. Again a few arrays are used to store enemy angles, distance (from the centre) and speed (enemyMove), however an additional variable (enemyHealth) has been added so that we can have different results from different weaponry.

07 Player and power-ups

Here we handle variables like the player angle, set score flags and define the power-up colour, fade and angle.

08 Main loop

This is where the meat is. To start with we're checking for key presses. We have it set to A and D (or the left and right keys) to move the player left and right, and then either the space bar or return key for firing. The fire button routine is fairly complex as we have to check to see if we have a laser available because we are restricted by maxLasers. We then check to see if they have reached the centre (radius2). If we have particle lasers enabled, we swap sides as each one is added so we have the dual barrel machine gun effect (very nice indeed):

```
// Check if Right cursor or the D key is
pressed
  IF scanKeyboard (scanRight) OR
scanKeyboard (scanD) THEN
   IF moveCount = 0 THEN
    oldPlayerAngle = playerAngle
    playerAngle = playerAngle + gap
```

```
    moveCount = moveDelay
    IF playerAngle > 360 THEN
     playerAngle = gap / 2
    ENDIF
   ENDIF
  ENDIF
 // Check if Left cursor or the A key is
pressed
  IF scanKeyboard (scanLeft) OR
scanKeyboard (scanA) THEN
   IF moveCount = 0 THEN
    oldPlayerAngle = playerAngle
    playerAngle = playerAngle - gap
    moveCount = moveDelay
    IF playerAngle < 0 THEN
     playerAngle = 360 - (gap / 2)
    ENDIF
   ENDIF
  ENDIF
 // Check if the Space bar or Return key is
pressed
  IF scanKeyboard (scanSpace) OR
scanKeyboard (scanReturn) THEN
   IF laserCount = 0 THEN
    IF numLasers < maxLasers THEN
     FOR i = 0 TO maxLasers CYCLE
      IF laserDist(i) = radius2 THEN
       laserDist(i) = SQRT (((centerX -
playerX) * (centerX - playerX)) + ((centerY
- playerY) * (centerY - playerY)))
       laserAngle(i) = playerAngle
       laserSide(i) = particleLaserSide
       IF particleLaserSide = 0 THEN
```

```
            particleLaserSide = 1
          ELSE
            particleLaserSide = 0
          ENDIF
          numLasers = numLasers + 1
          laserCount = laserReload
          BREAK
        ENDIF
      REPEAT
    ENDIF
  ENDIF
ENDIF
```

09 Release the power-up

We begin the game armed with a single-shot laser. We only release the power-up canister if we haven't already powered-up. This function is checked with IF particleLaser = FALSE THEN, after which a new one is released and all of its variables initialised.

10 Move the power-up

Next it is moved outward towards the player with powerupDist = powerupDist + (radius / 80) and then checked to see if it has arrived at the outer edge (radius).

11 Get the power-up

Then if so, is it in the same place as the player (playerAngle - gap / 2)? If it is then "Particle Laser!" is displayed and particleLaser =TRUE.

12 Adjust shot speed

This determines shot release frequency. Particle release (laserCount / 1.5) is significantly quicker than standard shot

release (laserCount / 1.05).

13 Top-up enemies

If we have fewer than the maximum number of enemies on-screen (maxEnemies) and we are within the enemyCount boundaries, then this if statement will introduce a new enemy.

```
// Check for enemies being present
  IF tempTime > 3000 THEN
   IF enemies = TRUE THEN
    IF enemyCount > 0 THEN
     enemyCount = enemyCount - 1
    ELSE
     IF enemyCount = 0 THEN
      IF numEnemies < maxEnemies THEN
       FOR i = 0 TO maxEnemies CYCLE
        IF enemyDist(i) = radius THEN
         enemyDist(i) = radius2
         enemyAngle(i) = (RND (vertices - 1)
* gap) + (gap / 2)
         enemyHealth(i) = 100
         numEnemies = numEnemies + 1
         enemyCount = enemyDelay
         BREAK
        ENDIF
       REPEAT
      ENDIF
     ENDIF
    ENDIF
   ENDIF
  ENDIF
```

14 Plot the stars

The star field routine runs through the maxStars variable, increasing the distance from the centre (starsDist) until it travels completely off the screen. At that point they're reset back to the middle with a random factor so they don't all appear dead in the centre. They are drawn with a simple PLOT (starX, starY) command.

```
// Routine to plot the stars
  COLOUR = White
  WHILE starNum < maxStars CYCLE
   starX = starsDist(starNum) * COS
(stars(starNum))
   starX = starsCenterX + starX
   starY = starsDist(starNum) * SIN
(stars(starNum))
   starY = starsCenterY + starY
   starsDist(starNum) = starsDist(starNum) +
starsSpeed(starNum)
   IF starX < 0 THEN
    starsDist(starNum) = RND (15) + 5
   ENDIF
   IF starX > gWidth THEN
    starsDist(starNum) = RND (15) + 5
   ENDIF
   IF starY < 0 THEN
    starsDist(starNum) = RND (15) + 5
   ENDIF
   IF starY > gHeight THEN
    starsDist(starNum) = RND (15) + 5
   ENDIF
   PLOT (starX, starY)
   starNum = starNum + 1
```

```
REPEAT
starNum = 0
```

15 Draw playing field

We explained this concept in detail last month and very little has changed. Basically, the playing field is drawn in segments around the circumference of a circle using just three LINE statements.

```
// Draw the playing field
  COLOUR = Blue
  WHILE angle < 360 CYCLE
   x = radius * COS (angle)
   x = centerX + x
   y = radius * SIN (angle)
   y = centerY + y
   x2 = radius2 * COS (angle)
   x2 = centerX2 + x2
   y2 = radius2 * SIN (angle)
   y2 = centerY2 + y2
   LINE (x, y, x2, y2)
   LINE (x, y, oldX, oldY)
   LINE (x2, y2, oldX2, oldY2)
   oldX = x
   oldY = y
   oldX2 = x2
   oldY2 = y2
   angle = angle + gap
REPEAT
```

16 Calculate and draw player

The player position is becoming more complex. The first stage introduces a new function: DEF FN lerp(a,b,c). This is

going to be used a lot from now on, so:

```
DEF FN lerp(a, b, c)
result = a + c * (b - a)
 = result
```

This takes two numbers, A and B, and interpolates between them so that C can be used as a distance or angle, or even a colour step. This enables us to work out a step in between two points on the screen and calculate an equal step between. This is then used to ensure smooth movement is made for any object anywhere on the screen, regardless of its size or location. Very clever indeed!

```
// Calculate the player position
 IF playerAngle - oldPlayerAngle < ((0 -
360) + gap) + 1 THEN
   tmpPlayerAngle = FN lerp(playerAngle +
360, oldPlayerAngle, moveCount / moveDelay)
  ELSE
   IF playerAngle - oldPlayerAngle > (360 -
gap) - 1 THEN
    tmpPlayerAngle = FN lerp(playerAngle,
oldPlayerAngle + 360, moveCount /
moveDelay)
   ELSE
    tmpPlayerAngle = FN lerp(playerAngle,
oldPlayerAngle, moveCount / moveDelay)
   ENDIF
  ENDIF
  playerX = (((radius * COS (tmpPlayerAngle
```

```
- (gap / 2))) + centerX) + ((radius * COS
(tmpPlayerAngle + (gap / 2))) + centerX)) /
2
  playerY = (((radius * SIN (tmpPlayerAngle
- (gap / 2))) + centerY) + ((radius * SIN
(tmpPlayerAngle + (gap / 2))) + centerY)) /
2
  angle = 0
  IF moveCount > 0 THEN
   moveCount = moveCount - 1
  ENDIF
```

17 Change perspective

Again covered in detail last month, this section shifts the playing field perspective depending on which corner of the screen the player is positioned.

18 Calculate and draw power-up

The IF powerupDist < radius THEN line makes sure the power-up exists as it has not yet reached the outer radius. Notice the use of the FN lerp function again, but this time to calculate the smooth gradient between two colours. See, we told you this would be a useful thing to learn to do! The colour evenly fades between random cycles. The power-up's distance from the centre is updated and is then drawn using a simple CIRCLE statement: CIRCLE (powerupX, powerupY, u, TRUE).

19 Calculate and draw lasers

While the actual movement of the lasers remains the same – apart from speed, that is – we need to calculate the laser positions depending on whether we are in single shot or particle mode. If in particle mode then both sides

need to be calculated. Finally, the lasers are drawn with a polyPlot function.

20 Check for laser hits

Each laser is checked to see if it enters the same airspace as an enemy and if so, the enemy is dealt with accordingly. At this stage all lasers have the same power so a single hit reduces their health by 100. Next month we'll be taking you through how to update this so that if an enemy is hit, it will be removed and the player's score will then increase by 10. Once a laser reaches the inner circle (radius2) then it is removed from the playing field.

21 Calculate and draw enemies

You should be getting familiar with the process now:

cycle through the number of items, in this case enemies (maxEnemies), check to see if they've reached the outer rim, and if not then use FN lerp to calculate a smooth movement step and apply it. EnemyDist is the distance from the inner circle (radius2) and enemyAngle is the direction it is heading. We use COS and SINE to work out the position and then a ployPlot function to draw the enemy.

22 Check for enemy hits

Next we test the enemy position against the player position and if they are the same, "Game Over" is displayed and the game ends… for now. Finally, we check the current angle of the enemy and make sure it is heading towards the player. Also the angle is tested and reset if it goes around the clock.

23 Calculate and draw player

This is rather simple now that everything else has been done. COS and SINE with U (the outer distance) again are used to determine the new angle and we finish off with a sequence of polyPlot commands to draw the player.

24 Display messages

The next block displays the score and any messages that might be in play, like "Particle Laser!" (more next month).

25 Check frame rate & recalibrate

This next section is huge but actually very straightforward. First off, check to see if we are below minfps and if so, recalibrate everything accordingly. All the key measures are reset for the new resolution so the inner and outer

radii are scaled to match the new size and so on.

The opposite happens if we are over maxfps, in that the resolution increases – if we go over maxxres then we keep it there, in this case the maximum resolution was set at the beginning at 1920 x 1080.

26 Main positional variables

Another long chunk but again very simple. This last but one block initialises the positions and values at the start of each game – this will become more important when we introduce level progression. The final block is the DEF FN lerp(a,b,c) function that we referred to earlier.

27 TBC...

And that's it for now! At this stage you have the basic shell of the game. Next month we will tidy everything up, introduce progressive scoring and levels, add awesome sounds, develop the difficulty settings, include a start-up screen and any other finishing touches. See you next month!

To find out more about FUZE BASIC and the FUZE in general, please visit **www.fuze.co.uk**

Embed Python in C

This month, we will learn how to use Python code within your usual C program to get the best of both worlds

Back in a previous issue of RasPi, we looked at how to call C functions from within a Python program to get more speed. But, there are times, within a C program, when you may want to execute some piece of Python code. Maybe you want to be able to run user code within your program, for example. This means you can enable users to use plug-ins to extend your program's functionality. The way we can do this is by embedding Python within the C program.

We will look at how to embed, how to run your Python code, and how to interact with the Python interpreter you've set up. This is functionality built into Python itself, so you don't need to install anything extra on your Raspberry Pi, aside from the development package for Python and GCC. You will need to install them with the command:

```
sudo apt-get install python-dev gcc
```

You should now have all of the tools you need to compile your code.

The first step is to start the interpreter. To access the functions you need, you will have to add the following line to

the head of your C source code file:

```
#include <Python.h>
```

You can now start to embed Python. The first function you need is void Py_Initialize(). The only other functions that can be called, before you initialise the interpreter, are Py_SetProgramName(), Py_SetPythonHome(), PyEval_InitThreads(), PyEval_ReleaseLock() and PyEval_AcquireLock(). Once this function finishes, you can start to interact with this interpreter. This starts up the interpreter, and loads the core modules __builtin__, __main__ and sys. But what about other modules? You can set the search path, where the interpreter will look to find modules, by using the function void Py_SetPythonHome(char *home). If you need the information, you can find the current module path with the function char* Py_GetPythonHome().

It does not set sys.argv, however. You need to use the function void PySys_ SetArgvEx(int argc, char **argv, int updatepath). This way, you can access any command line arguments that your Python code needs. You can check to see whether the interpreter is properly initialised by using the function int Py_IsInitialized(). It returns an integer for either true (nonzero) or false (zero). The simplest way to use your new interpreter is to use the function int PyRun_SimpleString(const char *command). This function takes a string that contains some arbitrary bit of code. If you have multiple lines of code that you want to run, you can use newline characters to separate lines. For example, you can print out the sine of an angle with:

```
PyRun_SimpleString("import  math\na  =  math.
```

```
sin(45)\nprint(‘The sine of 45 is ’ + a)”);
```

This function is a simplified version of int PyRun_SimpleStringFlags(const char *command, PyCompilerFlags *flags). This not only takes the command string, but also takes a struct of compiler flags for the Python compiler. You will need to check the development documentation online to see the details for these compiler flags.

Let’s say that you have a more complicated bit of code to execute. There are equivalent functions to work with Python script files. The simplified version is int PyRun_SimpleFile(FILE *fp, const char *filename). You actually hand in two references to your script. The first is a file handle that you get from the C function fopen() to open your script file, and the seconde is the name of the script you just opened. You will need to open your script file with the read permission. You also now need to worry about whether your program will have the correct file permissions on the file system to open this script. Proper coding means you should check this call to fopen() to verify that it completed and gave you a valid file handle. This simplified version doesn’t use any compiler flags, and closes the file handle after the function returns. The full version of the function is int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags). If closeit is true, then the file handle is closed. If the script is something you will want to run several times, set closeit to false so the file handle remains open. You can set any flags for the Python interpreter in the flags struct, similar to the PyRun_SimpleStringFlags() function call.

If this simple way of running code isn’t powerful enough, there are ways of interacting with the interpreter in a more direct fashion. The first step is learning how to send data back

and forth between the Python interpreter and the main body of your C program. The basic workflow is to convert your C variables to their Python equivalents, then call the Python functions you wish to use, and convert the Python results back into their equivalents within C. Python is an object-oriented language, so the core of communicating with the interpreter happens with the Py_Object construct. This provides the base for all the other types of objects you can use to communicate with Python. For example, create a Python string object with:

```
PyObject *pName;
pName = PyString_FromString("print('Hello World')");
```

You can then use this Python object when using Python functions. For example, if you stored the name of a Python module in the string pName, you could import it with the function call PyImport_Import(pName);. You can also get access to Python functions from your C code. You store a reference to the function in a PyObject, just as with data objects. The first step is to get the dictionary of the function names for the module in question with:

```
my_module = PyImport_AddModule("__main__");
my_dict = PyModule_GetDict(my_module);
```

Once you have the dictionary, you can get a reference to specific function with:

```
my_func = PyDict_GetItemString(my_dict, func_name);
```

where func_name is a string containing the function you want access to. You can then run the function with a command like

```
PyObject_CallObject(my_func, NULL);
```

With this access, you should be able to do just about anything you wish in Python.

Up to now, we have been looking at code interacting with the Python interpreter. But there are occasions when you want to allow the end user to have access to the interpreter. In these cases, you probably want to give your user access to a full Python console. You can do just such a thing with the function call Py_Main(argc, argv), where you hand in the argc and argv that you have from the C side of your program. This is fine for a console-based program, but for a GUI program, you need to create some kind of terminal window to allow the user to interact with the Python interpreter. This console will continue until the user explicitly quits from Python.

The last thing you need to do is to clean up after the interpreter. You can do this with the function void Py_Finalize(). The major issue with this function is that it destroys objects in a random order. If they depend on other objects, they may not be able to get cleaned up correctly. If you then try and re-initialise the interpreter again, it may fail due to an unclean finalisation step.

Now that you have your program written, you need to compile it. You need to include flags to tell the compiler where to find everything. Luckily, you can get these from Python itself. The flags needed for compiling are available with the command python-config --cflags. You also need to know where to find the libraries to link in, which are available with python-config --ldflags. Now, you have access to Python anywhere, even within another program.

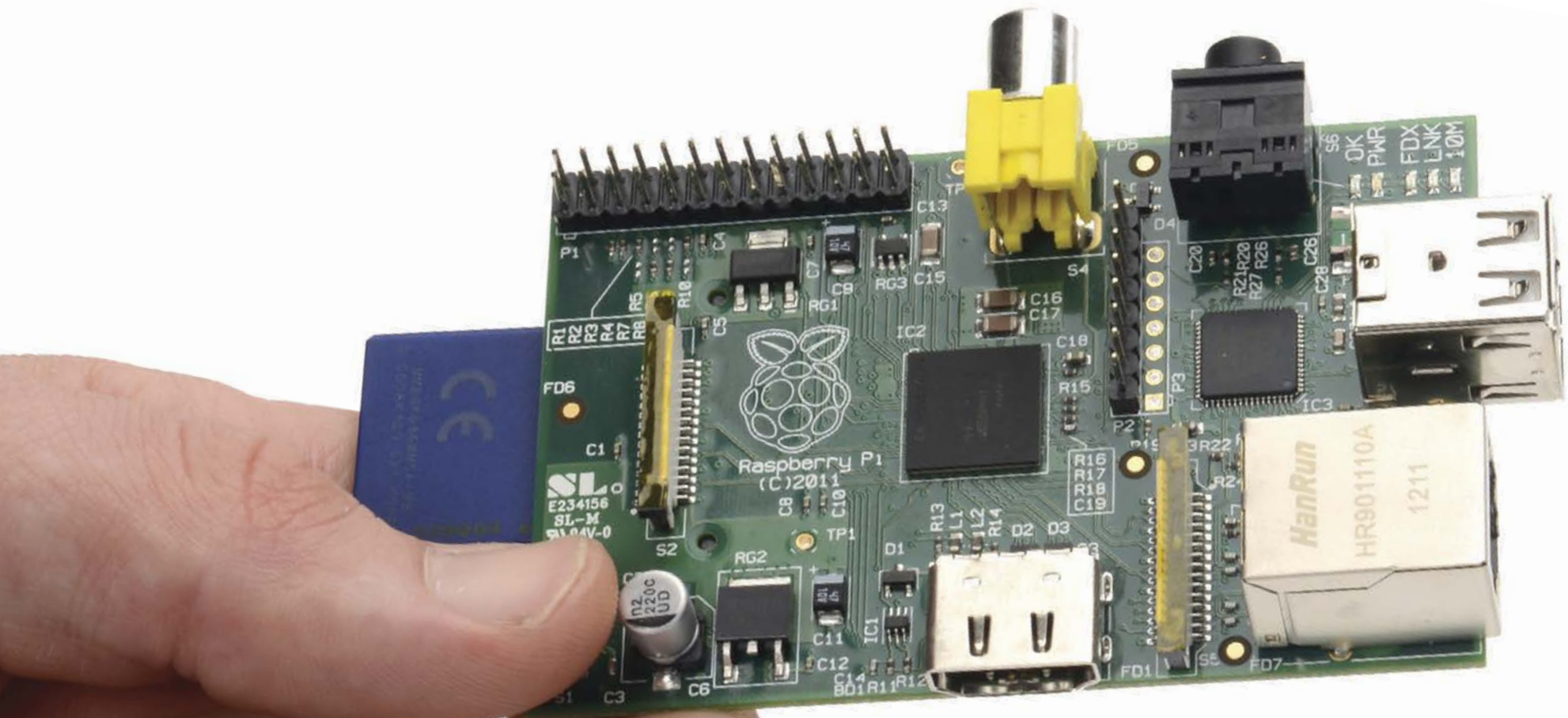
The Code LED MATRIX

```
# A simple way to run Python code
#include <Python.h>

int main(int argc, char *argv[]) {
   Py_SetProgramName(argv[0]);
   # Initialize the Python interpreter
   Py_Initialize();
   # Run your Python code
   PyRun_SimpleString("from time import time,ctime\n"
                      "print 'Today is',ctime(time())\n");
   # Don't forget to clean up
   Py_Finalize();
   return 0;
}
----------------------------------
# You can create an interactive Python console
#include <Python.h>

int main(int argc, char *argv[]) {
   Py_Initialize();
   Py_Main(argc, argv);
   Py_Finalize();
}
----------------------------------
# You can even run a script file
#include <Python.h>

int main(int argc, char *argv[]) {
   FILE *fp;
   Py_Initialize();
   fp = fopen("my_script.py", "r");
   PyRun_SimpleFile(fp, "my_script.py");
   Py_Finalize();
   fclose(fp);
}
```

Talking Pi

Join the conversation at...

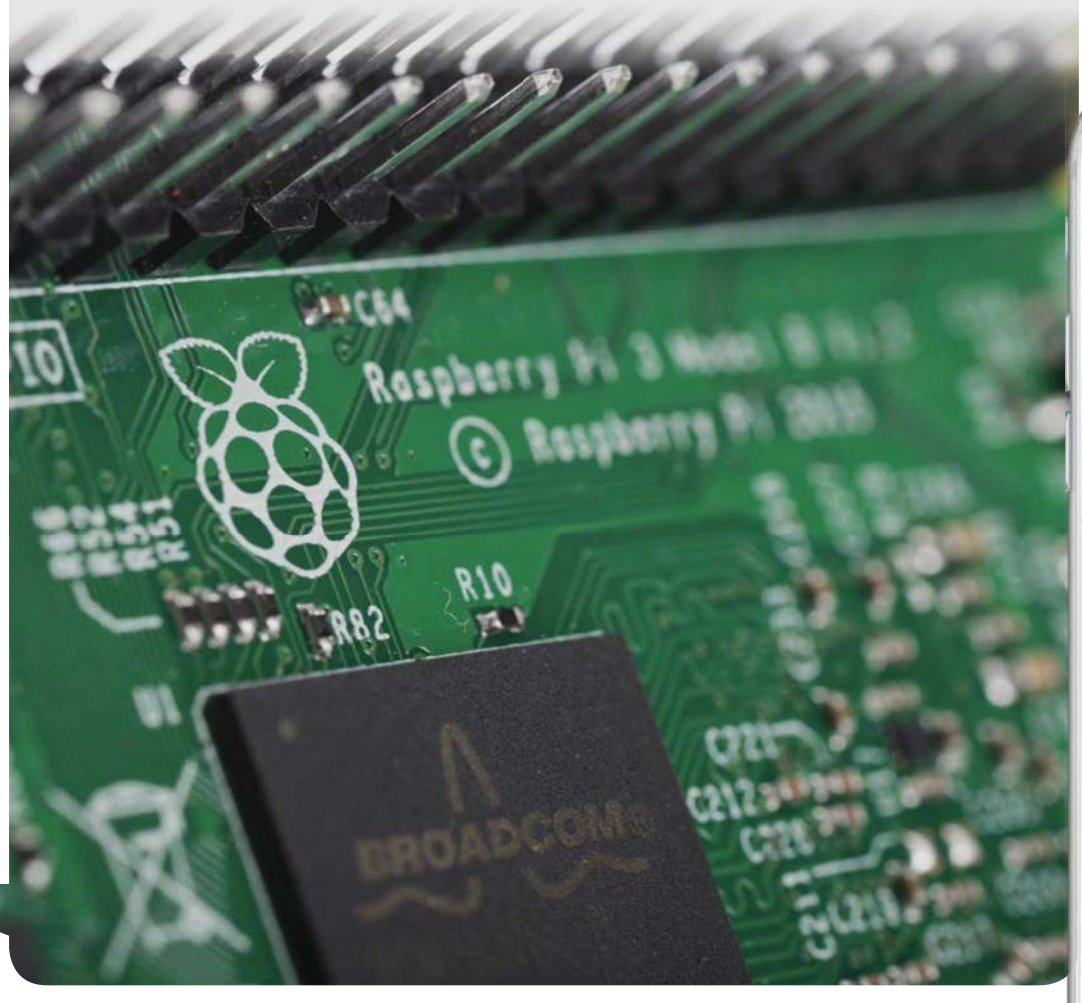
@linuxusermag | Linux User & Developer | @ RasPi@imagine-publishing.co.uk



The Raspberry Pi encourages a lot of hands-on work and this means it doesn't quite work like – or as easily as – your laser focus-tested smartphone interface, or even a normal computer. So we're answering your burning Raspberry Pi questions each issue – get in touch with us on Twitter, Facebook or by email.

Is Pixel a new operating system for the Raspberry Pi, and if so is it based on Linux or on something else instead?
Sam via email

Pixel isn't a completely new operating system – it's based on the original Raspbian, a variant of Debian forked for the Raspberry Pi. Pixel takes this further – its user interface has been designed specifically for the Pi. It's lightweight (so it won't take up too much in the way of the Pi's resources), and it's based on the X Windows System, which you may be familiar with from other distros. The name stands for Pi Improved Xwindows Environment, Lightweight (which is a bit of a mouthful) and it was designed by user experience engineer Simon Long, who began the project over two years ago. Now that it's reached fruition, why not install it?



Keep up with the latest Raspberry Pi news by following @LinuxUserMag on Twitter. Search for the hashtag #RasPiMag

JUST A SCORE
WHAT'S YOUR JUST A SCORE?

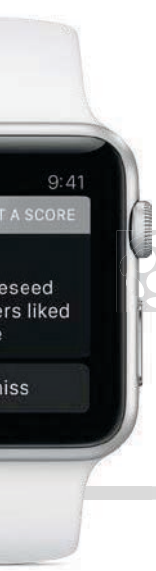
Have you heard of Just A Score? It's a new, completely free app that gives you all the latest review scores. You can score anything in the world, like and share scores, follow scorers for your favourite topics and much more. And it's really good fun!

I think I've broken my NOOBS SD card, what do I do to fix it?
Kai via email

Whether your SD card is physically broken or simply corrupted, there's a good chance that it's probably had it and that you'll need to get another one. You can pick them up from most gadget or camera shops and even supermarkets, but the key thing is to look for a high-rated SDHC card that's at least 8GB in size. Once you've got a new one you can put NOOBS back on it, although sadly it's likely that you've lost whatever data was on the old card.

Is it true that the Pi 2 turns off when you photograph it?
Mike via email

When we first heard this story we thought it was an urban myth too, and while we haven't actually tried it ourselves, there is a thread on the official Raspberry Pi forum all about it. Take a look at **http://bit.ly/2ejCnBC**. The theory is that one of the exposed chips on the board isn't shielded well enough and is sensitive to light, so when a strong camera flash goes off near it (the community found that it's a Xenon flash that sets it off rather than the LED flashes common on phones, or even the light from high-powered bike lamps), it stops it running, although the Pi is still on. Don't try this at home!



Next issue

Get inspired **Expert advice** **Easy-to-follow guides**

Take night photos with your Pi

Get this issue's source code at:

www.linuxuser.co.uk/raspicode